

Nucleus PLUS

Reference Manual



Copyright (c) 1999
Accelerated Technology, Inc.
720 Oak Circle Dr. E.
Mobile, AL 36609
(334) 661-5770



Related Documentation

Nucleus PLUS Internals, by Accelerated Technology, describes, in considerable detail, the implementation of the Nucleus PLUS kernel.

Style and Symbol Conventions

Program listings, program examples, filenames, menu items/buttons and interactive displays are each shown in a special font.

Program listings and program examples - *Courier New*

Filenames - *COURIER NEW, ALL CAPS*

Interactive Command Lines - ***Courier New, Bold***

Menu Items/Buttons – *Times New Roman Italic*

Trademarks

MS-DOS is a trademark of Microsoft Corporation

UNIX is a trademark of X/Open

IBM PC is a trademark of International Business Machines, Inc.

Additional Assistance

For additional assistance, please contact us at the following:

Accelerated Technology
720 Oak Circle Drive, East
Mobile, AL 36609
800-468-6853
334-661-5770
334-661-5788 (fax)
support@atinucleus.com
<http://www.atinucleus.com>

Copyright (©) 1999, All Rights Reserved.

Document Part Number : 001026-001

Last Revised: June 16, 1999





Contents

Chapter 1 - Introduction	1
About Nucleus PLUS	2
Real-Time Applications	2
Why Nucleus PLUS is Needed	2
Chapter 2 – Getting Started	5
Application Development	6
Installing Nucleus PLUS	6
How to Use Nucleus PLUS	7
Application Initialization	8
Target System Considerations	9
Configuration Options	9
System Initialization	10
Memory Usage	10
Execution Threads	13
Initialization	13
System Error	13
Scheduling Loop	13
Task	13
Signal Handler	13
User ISR	14
LISR	14
HISR	14
Chapter 3 – Task Control	15
Introduction	16
Task States	16
Preemption	16
Relinquish	16
Time Slicing	17
Dynamic Creation	17
Determinism	17
Stack Checking	17
Task Information	17
Priority	17



Function Reference.....	19
Chapter 4 – Dynamic Memory	39
Introduction	40
Suspension	40
Dynamic Creation	40
Determinism	40
Dynamic Memory Pool Information	41
Function Reference.....	41
Example Source Code	52
Chapter 5 – Partition Memory	57
Introduction	58
Suspension	58
Dynamic Creation.....	58
Determinism.....	58
Partition Information	59
Function Reference.....	59
Example Source Code	70
Chapter 6 - Mailboxes	75
Introduction	76
Suspension	76
Broadcast	76
Dynamic Creation.....	76
Determinism.....	76
Mailbox Information.....	77
Function Reference.....	77
Example Source Code	93
Chapter 7 -Queues	97
Introduction	98
Message Size.....	98
Suspension	98
Broadcast	98
Dynamic Creation.....	98
Determinism.....	98
Queue Information.....	99
Function Reference.....	99
Example Source Code	118
Chapter 8 - Pipes	123
Introduction	124
Message Size.....	124
Suspension	124
Broadcast	124
Dynamic Creation.....	124
Determinism.....	125
Pipe Information.....	125



Function Reference	125
Example Source Code	142
Chapter 9 - Semaphores	147
Introduction	148
Suspension	148
Deadlock	148
Priority Inversion	148
Dynamic Creation	149
Determinism	149
Semaphore Information	149
Function Reference	149
Example Source Code	162
Chapter 10 – Event Groups	167
Introduction	168
Suspension	168
Dynamic Creation	168
Determinism	168
Event Group Information	168
Function Reference	169
Sample Source Code	183
Chapter 11 - Signals	187
Introduction	188
Signal Handling Routine	188
Enable Signal Handling	188
Clearing Signals	188
Multiple Signals	188
Determinism	188
Function Reference	189
Sample Source Code	195
Chapter 12 - Timers	199
Introduction	200
Ticks	200
Margin of Error	200
Hardware Requirement	200
Continuous Clock	200
Task Timers	200
Application Timers	201
Re-Scheduling	201
Enable/Disable	201
Dynamic Creation	201
Determinism	201
Timer Information	202
Function Reference	202
Example Source Code	216



Chapter 13 - Interrupts	219
Introduction	220
Protection	220
Low-Level ISR	220
High-Level ISR	220
HISR Information	221
Interrupt Latency	221
Application Interrupt Lockout	221
Direct Vector Access	221
Function Reference	222
Managed ISRs	237
Unmanaged ISRs	239
Chapter 14 – System Diagnostics	241
Introduction	242
Error Management	242
System History	242
Version Information	242
License Information	242
Building the PLUS Library	242
Function Reference	243
Example Source Code	251
Chapter 15 – I/O Drivers	255
Introduction	256
Common Interface	256
Driver Contents	256
Protection	256
Suspension	256
Dynamic Creation	256
Driver Information	257
Function Reference	257
Implementing an I/O Driver	269
Actual Driver Requests	269
Initialization	269
Assign	270
Release	271
Input	271
Output	272
Status	272
Terminate	273
Driver Implementation	273
Example Driver	275
Chapter 16 – Demo Application	277
Example Overview	278
Example System	279



Appendix A –Nucleus PLUS Constants	285
Nucleus PLUS Constants (Alphabetical Listing).....	286
Nucleus PLUS Constants (Numerical Listing)	288
Appendix B – Error Conditions	291
Nucleus PLUS Fatal System Errors	292
Nucleus PLUS Error Codes	292
Appendix C - I/O Driver Request Structures	295
Nucleus PLUS I/O Driver Constants	296
Nucleus PLUS I/O Driver C Structures.....	296
Appendix D – Techniques For Conserving Memory.....	299
Data Initialization	300
NU_MAX_LISRS.....	300
TC_PRIORITIES.....	300
HISR Stack Sharing.....	301
TCD_Lowest_Set_Bit.....	301
Using a Smaller INT Option.....	301





1

Introduction

About Nucleus PLUS

Real Time Applications

Why Nucleus Plus Is Needed



About Nucleus PLUS

Nucleus PLUS is a real-time, preemptive, multitasking kernel designed for time-critical embedded applications. Approximately 95% of Nucleus PLUS is written in ANSI C. Because of this, Nucleus PLUS is extremely portable and is currently available for use with most microprocessor families.

Nucleus PLUS is usually implemented as a C library. Real-time Nucleus PLUS applications are linked with the Nucleus PLUS library. The resulting object may be downloaded to the target, or placed in ROM. Nucleus PLUS is normally delivered in source code form. Having access to the Nucleus PLUS source promotes greater understanding and permits application-specific modifications.

Real-Time Applications

What is real-time? Real-time is a term used to describe software that must produce the correct response to external and internal events at the proper time. Real-time may be categorized as either *hard* or *soft* real-time. In soft real-time, failure to produce the correct response at the correct time is undesirable. However, such a failure in hard real-time is catastrophic.

Today's real-time applications are often responsible for a variety of duties or tasks. Tasks typically have a single purpose, and are therefore implemented as semi-independent program segments. Most applications consist of hard and soft real-time tasks.

Why Nucleus PLUS is Needed

Due to inherent differences in task importance, the method used to share a processor between tasks is very important. Simple real-time applications, and usually those of a more soft real-time nature, might embed processor allocation logic inside the application tasks. Implementations of this kind typically take the form of a control loop that continually checks for tasks to execute. Such techniques suffer from the following problems:

Slow Response Time - The worst case time required to detect a critical event is the duration of the worst case thread of execution.

Modification Difficulties - Since processor allocation logic is dispersed throughout the application code, the time required for each task to execute is dependent on the processing time of other tasks. Therefore, a code change in a single task could result in the failure of the entire system.

Reduced Throughput - As the number of tasks increases, the amount of time wasted looking for tasks to execute increases. This time could be better spent doing something meaningful.

Difficult Software Development - Applications of this type typically have many interdependencies, making the coordination of multiple engineers more difficult. Additionally, porting such applications to other microprocessors may be more difficult.

Nucleus PLUS eliminates the need for processor allocation in the application software. When a more important task requires execution, Nucleus PLUS suspends the currently



executing task and starts the higher-priority task. After the higher-priority task finishes, the suspended task is resumed. The worst-case task response time under Nucleus PLUS is the amount of time required to suspend the executing task and resume the more important task. Nucleus PLUS provides quick and constant response time. Because of this, modifications, and even additions of completely new tasks can be made without affecting critical system response requirements.

Besides managing task execution, Nucleus PLUS also provides facilities that include task communication, task synchronization, timers, and memory management.

From the software development standpoint, Nucleus PLUS fosters less task interdependence and greater modularity. Because of this, multiple engineers may work on tasks without worrying about the side-effects present in non-Nucleus PLUS applications. Nucleus PLUS also provides a runtime environment that is completely independent of the target processor. This benefits the development effort in two ways: First, engineers may concentrate on the real-time application instead of the intricacies of the underlying processor; Second, engineers may develop applications that execute on most popular microprocessors.

To summarize, Nucleus PLUS greatly enhances the development of real-time applications. This translates into lower development costs and shorter development time. Since Nucleus PLUS allows easy migration of applications to new processor families, the application development investment is protected.





Getting Started

Application Development

Installing Nucleus PLUS

How To Use Nucleus PLUS

Application Initialization

Target System Considerations

Configuration Options

System Initialization

Memory Usage

Execution Threads

Application Development

Embedded, real-time applications are typically developed on what is called a host computer system. The IBM PC and UNIX workstation are good examples of host systems. Application software usually runs on a separate computer system, commonly called the target system. However, the IBM PC is an exception to this rule. It can serve as both a host and a target for Nucleus PLUS applications. Applications that run on the IBM PC take the form of an EXE file.

Building an embedded real-time application is fairly straightforward. Application software files, residing on the host system, are compiled/assembled into object form and linked together. The resulting image is either downloaded to the target system or placed in ROM on the target system.

Debugging software on the target system usually involves an In-Circuit Emulator (ICE) tool or a Target-Resident Monitor (TRM). Having an ICE tool is the better option, since an ICE tool gives engineers complete control and knowledge of the target system hardware. ICE tools are especially useful during the checkout of new hardware. Because of the cost, and in some cases the limited availability of ICE's, many projects use TRM's for debugging. A TRM is a small software component that runs on the target system (usually from ROM). TRM's provide services that include downloading, breakpoints, and memory access. Both ICE's and TRM's are controlled by the host system. This is often accomplished through a serial interface.

Source-level debugging allows engineers to debug an application using the actual C source code. This capability requires an additional program on the host system that makes associations between the C source code and what is in the target system memory. Most source-level debuggers use ICE's or TRM's to actually control and access the target system hardware.

Nucleus PLUS is integrated with numerous C source-level debuggers. In addition, the Nucleus PLUS Debugger is available to add extended multitasking debugging capabilities for Nucleus PLUS applications.

Installing Nucleus PLUS

The entire Nucleus PLUS system requires approximately 2 megabytes of disk space on the host system. Please see your port specific documentation for a full explanation of the installation process. Installation procedures vary from one target environment to another.



How to Use Nucleus PLUS

Nucleus PLUS is designed for use as a C library. Services used inside application software are extracted from the Nucleus PLUS library and combined with the application objects to produce the complete image. This image may be downloaded to the target system or placed in ROM on the target system.

The steps for using Nucleus PLUS are described in the following generic form:

- 1) Make changes, if necessary, to the low-level system initialization file, `INT`.
Note: this file is usually delivered in assembly language form and its extension is development tool specific.
- 2) Define the `Application_Initialize` function, which is executed by Nucleus PLUS prior to starting the system. Note the file `NUCLEUS.H` must be included in order to make Nucleus PLUS service calls.
- 3) Define application tasks. If Nucleus PLUS services are used, the file `NUCLEUS.H` must be included.
- 4) Compile and/or assemble all application software, including the low-level system initialization file `INT`.
- 5) Link `INT` and all application object files with the Nucleus PLUS library and any necessary development tool libraries.
- 6) Download the complete application image to the target system and let it run!

Please review the processor and development system documentation for additional information, including specific details on how to use the compiler, assembler, and linker.



Application Initialization

The `Application_Initialize` routine is responsible for defining the initial application environment. This includes tasks, mailboxes, queues, pipes, semaphores, event groups, memory pools, and other Nucleus PLUS objects.

`Application_Initialize` is provided with a pointer to the first available memory address. Memory after this address is not used by the compiler or Nucleus PLUS, and is therefore available to the application. Although the specific contents of `Application_Initialize` depend on the application, the following template is always valid:

```
#include <nucleus.h>
void Application_Initialize(void *first_available_memory)
{
    /* Application-specific initialization of Nucleus PLUS
       objects, including the creation of tasks, mailboxes,
       queues, pipes, event groups, and memory pools. */
}
```

Services called from the initialization routine cannot try to suspend, since the initialization routine does not execute as a task. Also, note that at least one task or interrupt handler must be created by `Application_Initialize`, and that `Application_Initialize` is the last routine to execute prior to execution of the first task.

The following example of `Application_Initialize` creates a memory pool and a task. Notice that this example does not check for any error conditions.

```
#include <nucleus.h>

/* Define task control structure. */
NU_TASK Task;

/* Define dynamic memory pool control structure. */
NU_MEMORY_POOL Memory_Pool;

void Application_Initialize(void *first_available_memory)
{
    void *stack_ptr;

    /* Create a 4,000 byte dynamic memory pool that starts at
       the first available address. */
    NU_Create_Memory_Pool(&Memory_Pool, "SYSTEM",
                        first_available_memory, 4000, 50,
                        NU_FIFO);

    /* Allocate the task's stack from the memory pool. */
    NU_Allocate_Memory(&Memory_Pool, &stack_ptr, 500,
                      NU_NO_SUSPEND);

    /* Create an application task with the function
       abc (0, NU_NULL) as the entry point. */
    NU_Create_Task (&Task, "ABC_TASK", abc, 0,
                  NU_NULL, stack_ptr, 500, 10, NU_PREEMPT,
                  NU_START);
}
```



Target System Considerations

The size of the Nucleus PLUS instruction area varies from a maximum of 20Kb on Complex Instruction Set Computer (CISC) architectures to roughly 40Kb on Reduced Instruction Set Computer (RISC) architectures. As for data structures, Nucleus PLUS requires a minimum of 1.5Kb of RAM. This does not include the amount of memory required for application tasks, queues, pipes, and other Nucleus PLUS objects.

Since Nucleus PLUS does not attempt to modify any preset data elements, it may easily be placed in ROM. Additionally, Nucleus PLUS is compatible with the ROM options available with each development environment.

If the target system contains a Target-Resident Monitor (TRM), the Nucleus PLUS application must be loaded to a memory area not used by the TRM. Additionally, the application must only take the interrupt vectors needed, since the TRM uses interrupt vectors to perform breakpoints and other functions.

Configuration Options

Nucleus PLUS applications have one conditional compilation option. By defining the `NU_NO_ERROR_CHECKING` symbol on the command line used to compile an application source file, all error checking logic in Nucleus PLUS services is bypassed. This results in improved Nucleus PLUS service performance.

There are several conditional compilation options available for use when building the Nucleus PLUS library. Generally, these options may be added to the compile commands inside the `BUILD_LI.BAT`, or `PLUS.BAT` file, which contains all of the commands necessary to build the Nucleus library. It is highly recommended that you refer to your target specific notes for the port that you are using for full details.

The conditional compilation symbols available and their corresponding meanings are defined as follows:

Compilation Symbol	Meaning
<code>NU_ENABLE_HISTORY</code>	Results in a history entry for each service call. This symbol may be added to commands for compilation of any or all <code>**C.C</code> files.
<code>NU_ENABLE_STACK_CHECK</code>	Enables stack checking. This symbol may be added to commands for compilation of any or all <code>**C.C</code> files.
<code>NU_ERROR_STRING</code>	Builds an ASCII error message if a fatal system error is encountered. This option is only applicable when compiling <code>ERD.C</code> , <code>ERI.C</code> , and <code>ERC.C</code> .
<code>NU_NO_ERROR_CHECKING</code>	Disables error checking on Nucleus PLUS system calls.

System Initialization

The `INT_INITIALIZE` routine is typically the first to execute in a Nucleus PLUS system. For most target environments, the hardware reset vector should contain the address of `INT_INITIALIZE`. `INT_INITIALIZE` is responsible for all target-dependent initialization. Target dependent initialization often includes setting up various processor control registers, the interrupt vector table, global C data elements, several Nucleus PLUS variables, and the system stack pointer. When `INT_INITIALIZE` is finished, control is transferred to the high-level Nucleus PLUS initialization routine `INC_INITIALIZE`. Note that control never returns to `INT_INITIALIZE`.

`INC_INITIALIZE` calls the initialization routines of each Nucleus PLUS component. After all Nucleus PLUS initialization is complete, `INC_INITIALIZE` calls the user-supplied initialization routine “`Application_Initialize`.”

The `Application_Initialize` routine is responsible for defining the initial application environment. Initial application tasks, mailboxes, queues, pipes, semaphores, event groups, memory pools, and other Nucleus PLUS objects are defined in this routine.

After `Application_Initialize` returns, `INC_Initialize` initiates task scheduling.

Memory Usage

Nucleus PLUS provides applications with the ability to designate memory utilization for each system object. System objects include tasks, HISRs, queues, pipes, mailboxes, semaphores, event flag groups, memory partition pools, dynamic memory pools, and I/O drivers. Each of the previously mentioned system objects requires a control structure. Some of the system objects require additional memory. For example, task creation requires memory for the control block and memory for the stack. All memory required by a system object is supplied during its creation.

Flexibility is the greatest benefit of this technique. For example, suppose a target board is equipped with a limited amount of high-speed memory. Performance of a high-priority task may be significantly increased by locating its task control block and stack in this high-speed memory area. Other tasks in the system may use a more abundant, but slower memory area. Of course, the performance of other system objects can be improved in a similar manner.

There are several ways to allocate memory for system objects. The easiest method is to allocate the memory using global C data structures. Another method is to dynamically allocate the memory, either from a dynamic memory pool or a partition memory pool. The third method is to allocate the memory from absolute physical areas on the target system.



Allocating memory for system objects using global C data structures is the easiest method for allocating control structures. The following are examples of control block allocation for each type of system object:

System Object	Example
NU_TASK	Example_Task;
NU_HISR	Example_HISR;
NU_DRIVER	Example_Driver;
NU_QUEUE	Example_Queue;
NU_MAILBOX	Example_Mailbox;
NU_PIPE	Example_Pipe;
NU_SEMAPHORE	Example_Semaphore;
NU_EVENT_GROUP	Example_Event_Group;
NU_PARTITION_POOL	Example_Partition_Pool;
NU_MEMORY_POOL	Example_Memory_Pool;

Example_* are control blocks that reside in the global C data area. A pointer to the appropriate control block is passed to the appropriate create service. Stacks, queue areas, memory pool areas, and other system object areas may also be allocated as global C data structures, however it is generally less attractive than subsequent methods. **Note:** local C data structure allocation is also legal, providing that the objects defined within a function are no longer in use when the function returns.

Allocating memory for system objects from a Nucleus PLUS memory pool is quite common. Memory pools are themselves system objects, and therefore may be created to manage various memory areas.

The following is an example of allocating a task control block and a 1000-byte stack from a previously created dynamic memory pool (System_Memory is the global C control block of this previously created memory pool) :

```

NU_TASK    *Example_Task_Ptr;
VOID       *Example_Stack_Ptr;
.
.
.
/* Allocate memory for task control structure. */
NU_Allocate_Memory(&System_Memory, (VOID **)&Example_Task_Ptr,
                  sizeof(NU_TASK), NU_NO_SUSPEND);

/* Allocate memory for task stack. */
NU_Allocate_Memory(&System_Memory, &Example_Stack_Ptr,
                  1000, NU_NO_SUSPEND);

/* Task create call is supplied with Example_Task_Ptr and the
   Example_Stack_Ptr. */

```

Finally, the last type of system object memory allocation involves specific memory areas on the target board. Assume that address 0x200000 is a high-speed memory area of 4096 bytes. The first example creates a dynamic memory pool in that memory area.

The second example allocates memory for a high-priority task, with both the control block and a 2000-byte stack in the high-speed memory area.

Example 1:

```
NU_MEMORY_POOL    System_Memory;
.
.
.
/* Create a dynamic memory pool that manages the high-speed
   memory at 0x200000. */
NU_Create_Memory_Pool(&System_Memory, "SYSMEM", (VOID *)
0x200000, 4096, 20, NU_FIFO);
```

Example 2:

```
NU_TASK          *Example_Task_Ptr;
VOID              *Example_Stack_Ptr;
CHAR              *High_Speed_Mem_Ptr;
.
.
/* Put starting address into high-speed memory pointer. */
High_Speed_Mem_Ptr = (CHAR *) 0x200000;

/* Allocate the task control block at beginning. */
Example_Task_Ptr = (NU_TASK *) High_Speed_Mem_Ptr;

/* Adjust the high-speed memory pointer. */
High_Speed_Mem_Ptr = High_Speed_Mem_Ptr + sizeof(NU_TASK);

/* Allocate the task stack area. */
Example_Stack_Ptr = (VOID *) High_Speed_Mem_Ptr;

/* Adjust the pointer to the high-speed memory area in case more
allocation is needed. */
High_Speed_Mem_Ptr = High_Speed_Mem_Ptr + 2000;

/* Call create task with Example_Task_Ptr & Example_Stack_Ptr. */
```



Execution Threads

A Nucleus PLUS application is always in one of eight possible threads of execution. The following is a list of all possible execution threads:

- Initialization
- System Error
- Scheduling Loop
- Task
- Signal Handler
- User ISR
- LISR
- HISR

Initialization

The initialization thread is the first thread of execution in the system. The entry point of the initialization thread is `INT_Initialize`. After the `Application_Initialize` function returns, the initialization thread is terminated by transferring control to the scheduling loop.

System Error

There are several possible system errors, most of which are detected during initialization. However, stack overflow conditions are detected during task and HISR execution. This thread of execution starts when the function `ERC_System_Error` is called. By default, system errors are fatal and therefore control stays in this thread. See Appendix B for system error codes.

Scheduling Loop

Entry to the scheduling loop occurs at `TCT_Schedule`. This thread of execution is responsible for transferring control to the highest priority HISR or task ready for execution. While there are no tasks or HISRs ready to execute, control stays in a simple loop within `TCT_Schedule`.

Task

Task threads represent the majority of application processing threads. Each task thread has its own stack. The entry of each task thread is specified during task creation. Task threads have full access to Nucleus PLUS services.

Signal Handler

A signal handler thread executes on top of the associated task's thread. Signal handler threads have limited access to Nucleus PLUS services. The primary limitation is that self-suspension is not allowed.



User ISR

User Interrupt Service Routine threads are typically small assembly language routines that are tied directly to an interrupt vector. Such threads are responsible for saving and restoring any registers used. Nucleus PLUS services are completely off-limits to this type of thread. In fact, C functions are also off-limits, unless the thread saves and restores all registers used by the compiler.

LISR

Low-Level Interrupt Service Routines are registered with Nucleus PLUS. This allows Nucleus PLUS to save and restore all necessary registers. LISR threads may therefore be written in C. LISR threads have limited access to Nucleus PLUS services; the most important is the activate-HISR service.

The following services are available from LISRs:

```
NU_Activate_HISR
NU_Local_Control_Interrupts
NU_Current_HISR_Pointer
NU_Current_Task_Pointer
NU_License_Information
NU_Retrieve_Clock
```

HISR

High-Level Interrupt Service Routines form the second part of a Nucleus PLUS interrupt. HISR threads are scheduled in a manner similar to task threads, and also may call most of the Nucleus PLUS services. However, HISR threads are not allowed to make any self-suspension requests. The entry point of an HISR routine is determined during HISR creation.



3

Task Control

Introduction

Task States

Function Reference



Introduction

A task is a semi-independent program segment with a dedicated purpose. Most modern real-time applications require multiple tasks. Additionally, these tasks often have varying degrees of importance. Managing the execution of competing, real-time tasks is the main purpose of Nucleus PLUS.

Task States

Each task is always in one of five states: *executing*, *ready*, *suspended*, *terminated*, or *finished*. The following list describes each of the task states:

State	Meaning
executing	Task is currently running.
ready	Task is ready, but another task is currently running.
suspended	Task is dormant while waiting for the completion of a service request. When the request is complete, the task is moved to the ready state.
terminated	Task was killed. Once in this state, the task cannot execute again until it is reset.
finished	Task finished its processing and returned from initial entry routine. Once in this state, the task cannot execute again until it is reset.

Preemption

Preemption is the act of suspending a lower priority task when a higher priority task becomes ready. For example, suppose a task with a priority of 100 is executing. If an interrupt occurs that readies a task with a priority of 20, the task with priority 20 is executed before the interrupted task is resumed. Preemption also occurs when a lower priority task calls a Nucleus PLUS service that makes a higher priority task ready.

Preemption may be disabled on an individual task basis. When preemption is disabled, no other task is allowed to run until the executing task suspends, relinquishes control, or enables preemption. A task that suspends or relinquishes control with preemption disabled has preemption disabled when it is resumed.

A task is created with preemption either enabled or disabled. Preemption may also be enabled and disabled during task execution.

Relinquish

A mechanism is provided to share the processor with other ready tasks at the same priority level in a round-robin fashion. When a task requests this service, all other ready tasks at the same priority are executed before the originally executing task is resumed.



Time Slicing

Time slicing provides another mechanism to share the processor with tasks having the same priority. A time slice corresponds to the maximum number of timer ticks (timer interrupts) that can occur before all other ready tasks at the same priority level are given a chance to execute. A time-slice behaves like an unsolicited task relinquish. Note that disabling preemption also disables time slicing.

Dynamic Creation

Nucleus PLUS tasks are created and deleted dynamically. There is no preset limit on the number of tasks an application may have. Each task requires a control block and a stack. The memory for each element is supplied by the application.

Determinism

Processing time associated with suspending and resuming tasks is a constant. It is not affected by the number of application tasks. Additionally, the method in which tasks execute is not only predictable, but also guaranteed. Higher priority, ready tasks execute before lower priority, ready tasks. Ready tasks of the same priority execute in the order they became ready.

Stack Checking

Application tasks may check the amount of memory left on the current stack. This function also keeps track of maximum stack usage. Stack checking may also be enabled inside Nucleus PLUS services through a conditional compilation option.

Task Information

Application tasks may obtain a list of active tasks. Detailed information about each task can also be obtained. This information includes the task name, current state, scheduled count, priority, and stack parameters.

Priority

A task's priority is defined during task creation. Additionally, dynamic modification of a task's priority is supported. A task that has a numerical priority of 0 has a higher priority than a task with a numerical priority of 255. Nucleus PLUS executes higher priority tasks before lower priority tasks. Tasks having the same priority are executed in the order in which they became ready for execution.

Note: Care must be taken when assigning priorities to application tasks. If care is not taken, the priorities can cause task *starvation* and excessive system overhead.

A task may only execute if it is the highest priority, ready task. Therefore, if a task or tasks at a certain priority are always ready, all tasks of a lower priority never execute. This situation is called starvation. There are several cures for this. First, higher priority tasks should suspend to allow lower priority tasks to execute.



Tasks that run at or near continuously should have a relatively low priority. Another technique to combat starvation is to gradually raise the priority of the starving task.

A substantial amount of additional overhead may be incurred if task priorities are used improperly. Consider a system of three tasks named A, B, and C. Each task has similar processing that consists of waiting for a message and/or sending a message in an infinite loop. Task A waits for a message from an Interrupt Service Routine (ISR) and then sends a message to task B. Task B waits for a message from task A and then sends a message to task C. Task C waits for a message from task B and then increments a counter. After this simple system starts (regardless of priority), all tasks execute briefly, and then suspend waiting for a message.

If all of the tasks have the same priority, the following set of events take place after the ISR sends a message to task A:

- Task A is resumed
- Task A sends a message to task B, making task B ready
- Task A suspends waiting for another message
- Task B is resumed
- Task B sends a message to task C, making task C ready
- Task B suspends waiting for another message
- Task C is resumed
- Task C increments a counter
- Task C suspends waiting for another message

Now assume that task A is lower priority than task B and task B is lower priority than task C. The following events take place after the ISR sends a message to task A:

- Task A is resumed
- Task A sends a message to task B, making task B ready
- Task A relinquishes to higher priority task B
- Task B is resumed
- Task B sends a message to task C, making task C ready
- Task B relinquishes to higher priority task C
- Task C is resumed
- Task C increments a counter
- Task C suspends waiting for another message
- Task B is resumed again
- Task B suspends waiting for another message
- Task A is resumed again
- Task A suspends waiting for another message

The application work performed in both of the previous examples is the same, i.e. two tasks sent messages and three tasks received messages. However, the amount of system overhead in resuming and suspending tasks doubled. Also notice the delay incurred in task A between sending a message and waiting for another message in the last example.

Obviously the previous example systems are useful only to show how priorities can affect system overhead. Different priorities are necessary for real-time applications to respond to external events and to allocate processing time to relatively more-important tasks. However, in order to reduce unnecessary system overhead, the number of different priorities in an application should be minimized.



Function Reference

The following function reference contains all Nucleus PLUS task control services. The following functions are contained in this reference:

- NU_Change_Preemption
- NU_Change_Priority
- NU_Change_Time_Slice
- NU_Check_Stack
- NU_Create_Task
- NU_Current_Task_Pointer
- NU_Delete_Task
- NU_Established_Tasks
- NU_Relinquish
- NU_Reset_Task
- NU_Resume_Task
- NU_Sleep
- NU_Suspend_Task
- NU_Task_Information
- NU_Task_Pointers
- NU_Terminate_Task



NU_Change_Preemption

```
OPTION NU_Change_Preemption(OPTION preempt)
```

This service changes the preemption posture of the currently executing task. If the preempt parameter contains `NU_NO_PREEMPT`, preemption of the calling task is disabled. Otherwise, if the preempt parameter contains `NU_PREEMPT`, preemption of the calling task is enabled. **Note:** disabling preemption also disables any time-slice associated with the calling task.

Overview

Option	
Tasking Changes	No
Allowed From	Task
Category	Task Control Services

Parameters

Parameter	Meaning
preempt	Valid options for this parameter are <code>NU_PREEMPT</code> and <code>NU_NO_PREEMPT</code> . <code>NU_PREEMPT</code> indicates that the task is preemptable, while <code>NU_NO_PREEMPT</code> indicates that the task is not preemptable. Note: time slicing is disabled if the task is not preemptable.

Return Value

The previous preemption posture (either `NU_NO_PREEMPT` or `NU_PREEMPT`) is returned.

Example

```
OPTION      old_preempt;
/* Disable preemption of the current task. */
old_preempt = NU_Change_Preemption(NU_NO_PREEMPT);
.
.
.
/* Restore previous preemption posture. */
NU_Change_Preemption(old_preempt);
```

See Also

`NU_Create_Task`, `NU_Change_Priority`, `NU_Change_Time_Slice`



NU_Change_Priority

```
OPTION NU_Change_Priority(NU_TASK *task,
                          OPTION new_priority)
```

This service changes the priority of the specified task to the value contained in `new_priority`. Priorities are numerical values ranging from 0 to 255. Lower numerical values indicate greater task priority.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Control Services

Parameters

Parameter	Meaning
<code>task</code>	Pointer to the user-supplied task control block. Note: all subsequent requests made to this task require this pointer.
<code>new_priority</code>	Specifies a priority value between 0 and 255. The lower the numeric value, the higher the task's priority.

Return Value

This service returns the previous priority to the caller.

Example

```
NU_TASK      Task;
OPTION       old_priority;
.
.
.
/* Change the priority of the task control block "Task"
   to priority 10. Assume "Task" has previously been
   created with the Nucleus PLUS NU_Create_Task service call. */
old_priority = NU_Change_Priority(&Task, 10);
```

See Also

`NU_Create_Task`, `NU_Change_Preemption`, `NU_Change_Time_Slice`



NU_Change_Time_Slice

```
UNSIGNED NU_Change_Time_Slice(NU_TASK *task,
                               UNSIGNED time_slice)
```

This service changes the time slice of the specified task to the value contained in `time_slice`. If `time_slice` contains a value of 0, time slicing for the task is disabled.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Control Services

Parameters

Parameter	Meaning
<code>task</code>	Pointer to the user-supplied task control block. Note: all subsequent requests made to this task require this pointer.
<code>time_slice</code>	Indicates the maximum amount of timer ticks that can expire while executing this task. A value of zero in this field disables time slicing for this task.

Return Value

This service returns the previous time slice value to the caller.

Example

```
NU_TASK      Task;
UNSIGNED     old_time_slice;
.
.
.
/* Change the time slice of the task control block "Task" to
   35 timer ticks. Assume "Task" has previously been created
   with the Nucleus PLUS NU_Create_Task service call.*/
old_time_slice = NU_Change_Time_Slice(&Task, 35);
```

See Also

`NU_Create_Task`, `NU_Change_Priority`, `NU_Change_Preemption`

NU_Check_Stack

UNSIGNED NU_Check_Stack(VOID)

This service examines the stack usage of the caller. If the remaining amount of space is less than that required to save the caller's context, a stack overflow condition is present and control will not return to the caller. If a stack overflow condition is not present, the service returns the number of free bytes remaining in the stack. Additionally, this service keeps track of the minimum amount of available stack space.

Overview

Option	
Tasking Changes	No
Allowed From	HISR, Signal Handler, Task
Category	Task Control Services

Parameters

None

Return Value

This service returns the number of bytes currently available on the caller's stack.

Example

```
UNSIGNED remaining;

/* Check the current stack for an overflow condition.
Store the number of free stack bytes in "remaining." */
remaining = NU_Check_Stack( );
```

See Also

NU_Create_Task, NU_Create_HISR



NU_Create_Task

```
STATUS NU_Create_Task(NU_TASK *task, CHAR *name, VOID
                      (*task_entry)(UNSIGNED, VOID *),
                      UNSIGNED argc, VOID *argv,
                      VOID *stack_address, UNSIGNED stack_size,
                      OPTION priority, UNSIGNED time_slice,
                      OPTION preempt, OPTION auto_start)
```

This service creates an application task.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application Initialize, HISR, Signal Handler, Task
Category	Task Control Services

Parameters

Parameter	Meaning
task	Pointer to the user-supplied task control block. Note: all subsequent requests made to this task require this pointer.
name	Pointer to an 8 character name for the task. The name does not have to be null-terminated.
task_entry	Specifies the entry function of the task.
argc	An UNSIGNED data element that may be used to pass initial information to the task.
argv	A pointer that may be used to pass information to the task.
stack_address	Designates the starting memory location of the task's stack.
stack_size	Specifies the number of bytes in the stack.
priority	Specifies a priority value between 0 and 255. The lower the numeric value, the higher the task's priority.
time_slice	Indicates the maximum amount of timer ticks that can expire while executing this task. A value of zero in this field disables time slicing for this task.
preempt	Valid options for this parameter are NU_PREEMPT and NU_NO_PREEMPT. NU_PREEMPT indicates that the task is preemptable, while NU_NO_PREEMPT indicates that the task is not preemptable. Note: time slicing is disabled if the task is not preemptable.
auto_start	Valid options for this parameter are NU_START and NU_NO_START. NU_START places the task in a ready state after it is created. Tasks created with NU_NO_START must be resumed at a later time.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_TASK	Indicates the task control block pointer is NULL.
NU_INVALID_ENTRY	Indicates the task's entry function pointer is NULL.
NU_INVALID_MEMORY	Indicates the memory area specified by the stack_address is NULL.
NU_INVALID_SIZE	Indicates the specified stack size is not large enough.
NU_INVALID_PRIORITY	Indicates priority is invalid.
NU_INVALID_PREEMPT	Indicates that the preempt parameter is invalid. This error also occurs if a time slice is specified along with the no-preemption option.
NU_INVALID_START	Indicates the auto_start parameter is invalid.

Example

```

/* Assume task control block "Task" is defined as global
   data structure. This is one of several ways to allocate
   a control block. */

NU_TASK    Task;
STATUS     status;      /* Task creation status */
.
.

/* Create a task whose entry point is the function "task_entry"
   and that has a 2000-byte stack pointed to by "stack_ptr."
   Note the following additional parameters:
       argc and argv (0, NULL)
       priority is 200
       15 timer-tick time slice
       preemptable
       automatic start */

status = NU_Create_Task(&Task, "any name", task_entry, 0, NULL,
                       stack_ptr, 2000, 200, 15, NU_PREEMPT,
                       NU_START);

/* At this point status indicates if the service was successful. */

```

See Also

```

NU_Delete_Task, NU_Established_Tasks, NU_Task_Pointers,
NU_Task_Information, NU_Reset_Task

```



NU_Current_Task_Pointer

```
NU_TASK *NU_Current_Task_Pointer(VOID)
```

This service returns the currently active task pointer. If no task is currently active, an `NU_NULL` is returned. If a HISR is the active thread, and a task that could resume after the HISR completes has been interrupted, the return value is still `NU_NULL`.

Overview

Option	
Tasking Changes	No
Allowed From	LISR, Signal Handler, Task
Category	Task Control Services

Parameters

None

Return Value

This service returns a pointer to the currently active task control block.

Example

```
NU_TASK *task_ptr;

/* Obtain the currently active task's pointer. */
task_ptr = NU_Current_Task_Pointer( );
```

See Also

`NU_Established_Tasks`, `NU_Task_Pointers`, `NU_Task_Information`

NU_Delete_Task

```
STATUS NU_Delete_Task(NU_TASK *task)
```

This service deletes a previously created application task. The parameter `task` identifies the task to delete. Note that the specified task must be either in a finished or terminated state prior to calling this service. Additionally, the application must prevent the use of this task during and after deletion.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Control Services

Parameters

Parameter	Meaning
<code>task</code>	Pointer to the user-supplied task control block. Note: all subsequent requests made to this task require this pointer.

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_TASK</code>	Indicates the task pointer is invalid.
<code>NU_INVALID_DELETE</code>	Indicates the task is not in a finished or terminated state.

Example

```
NU_TASK    Task;
STATUS     status
.
.
/* Delete the task control block "Task". Assume "Task" has
   previously been created with the Nucleus PLUS NU_Create_Task
   service call. */
status = NU_Delete_Task(&Task);

/* At this point, status indicates whether the
   service request was successful. */
```

See Also

`NU_Create_Task`, `NU_Established_Tasks`, `NU_Task_Pointers`,
`NU_Task_Information`, `NU_Reset_Task`



NU_Established_Tasks

UNSIGNED NU_Established_Tasks(VOID)

This service returns the number of established application tasks. All created tasks are considered established. Deleted tasks are no longer considered established.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Control Services

Parameters

None

Return Value

This service call returns the number of established application tasks.

Example

```

UNSIGNED  total_tasks;

/* Obtain the total number of tasks. */
total_tasks = NU_Established_Tasks( );

```

See Also

NU_Create_Task, NU_Delete_Task, NU_Task_Pointers,
 NU_Task_Information, NU_Reset_Task



NU_Relinquish

VOID NU_Relinquish(VOID)

This service allows all other ready tasks of the same priority a chance to execute before the calling task runs again.

Overview

Option	
Tasking Changes	Yes
Allowed From	Task
Category	Task Control Services

Parameters

None

Return Value

None

Example

```

/* Allow other tasks that are ready at the same
priority to execute before the calling task
resumes. */
NU_Relinquish( );

```

See Also

NU_Sleep, NU_Suspend_Task, NU_Resume_Task, NU_Terminate_Task,
 NU_Reset_Task, NU_Task_Information

NU_Reset_Task

```
STATUS NU_Reset_Task(NU_TASK *task, UNSIGNED argc, VOID *argv)
```

This service resets a previously terminated or finished task. **Note:** this service does not resume the task after it is reset. `NU_Resume_Task` must be called to actually start the task again. The parameters of this service are further defined as follows:

Overview

Option	
Tasking Changes	No
Allowed From	HISR, Signal Handler, Task
Category	Task Control Services

Parameters

Parameter	Meaning
task	Pointer to the task control block.
argc	An UNSIGNED data element that may be used to pass information to the task.
argv	A pointer that may be used to pass information to the task.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_TASK	Indicates the task pointer is invalid.
NU_NOT_TERMINATED	Indicates the specified task is not in a terminated or finished state. Only tasks in a terminated or finished state can be reset.

Example

```
NU_TASK    Task;
STATUS     status
.
.
.
/* Reset the previously terminated task control block "Task".
Pass the task values of 0 and NULL for argc and argv. Assume
"Task" has previously been created with the Nucleus PLUS
NU_Create_Task service call. */
status = NU_Reset_Task(&Task, 0, NULL);
```

See Also

`NU_Create_Task`, `NU_Delete_Task`, `NU_Terminate_Task`, `NU_Resume_Task`,
`NU_Suspend_Task`, `NU_Task_Information`



NU_Resume_Task

```
STATUS NU_Resume_Task(NU_TASK *task)
```

This service resumes a task that was previously suspended by the `NU_Suspend_Task` service. Additionally, this service initiates a task that was previously reset or created without an automatic start.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Control Services

Parameters

Parameter	Meaning
task	Pointer to the user-supplied task control block.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_TASK	Indicates the task pointer is invalid.
NU_INVALID_RESUME	Indicates the specified task is not in an unconditionally suspended state.

Example

```
NU_TASK      Task;
STATUS       status
.
.
.
/* Resume the task control block "Task". Assume "Task"
   has previously been created with the Nucleus PLUS
   NU_Create_Task service call. */
status = NU_Resume_Task(&Task,);
```

See Also

`NU_Create_Task`, `NU_Suspend_Task`, `NU_Reset_Task`, `NU_Task_Information`



NU_Sleep

VOID NU_Sleep(UNSIGNED ticks)

This service suspends the calling task for the specified number of timer ticks.

Overview

Option	
Tasking Changes	Yes
Allowed From	Task
Category	Task Control Services

Parameters

Parameter	Meaning
ticks	Number of timer ticks that the task will be suspended.

Return Value

None

Example

```
/* Sleep for 20 timer ticks */  
NU_Sleep(20);
```

See Also

NU_Relinquish



NU_Suspend_Task

```
STATUS NU_Suspend_Task(NU_TASK *task)
```

This task unconditionally suspends the task specified by the pointer `task`. If the task is already in a suspended state, this service insures that the task stays suspended even after its original cause for suspension is lifted. `NU_Resume_Task` must be used to resume a task suspended in this manner.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Control Services

Parameters

Parameter	Meaning
<code>task</code>	Pointer to the user-supplied task control block. Note: all subsequent requests made to this task require this pointer.

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_TASK</code>	Indicates the task pointer is invalid.

Example

```
NU_TASK    Task;
STATUS     status;
.
.

/* Unconditionally suspend the task control block
   "Task". Assume "Task" has previously been created
   with Nucleus PLUS NU_Create_Task service call. */
status = NU_Suspend_Task(&Task);
```

See Also

`NU_Resume_Task`, `NU_Terminate_Task`, `NU_Reset_Task`



NU_Task_Information

```
STATUS NU_Task_Information(NU_TASK *task, CHAR *name,
                          DATA_ELEMENT *task_status,
                          UNSIGNED *scheduled_count,
                          OPTION *priority, OPTION *preempt,
                          UNSIGNED *time_slice,
                          VOID **stack_base,
                          UNSIGNED *stack_size,
                          UNSIGNED *minimum_stack);
```

This service returns various information about the specified task.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Control Services

Parameters

Parameter	Meaning
task	Pointer to the task.
name	Pointer to an 8 character destination area for the task's name.
task_status	Pointer to a variable to hold the current status of the task.
scheduled_count	Pointer to a variable to hold the number of times the task has been scheduled.
priority	Pointer to a variable to hold the task's priority.
preempt	Pointer to a variable to hold the task's preempt option. NU_PREEMPT indicates the task is preemptable, while NU_NO_PREEMPT indicates the task is not preemptable.
time_slice	Pointer to a variable to hold the task's time slice value. A value of zero indicates that time slicing for this task is disabled.
stack_base	Pointer to a memory pointer to hold the starting address of the task's stack.
size	Pointer to a variable to hold the total number of bytes in the task's stack.
minimum_stack	Pointer to a variable to hold the minimum amount of bytes left in the task's stack.

Task Status

The following table summarizes the possible values for the `task_status` parameter.

Parameter Value	Task Status
NU_READY	Ready to execute.
NU_PURE_SUSPEND	Unconditionally suspended.
NU_FINISHED	Returned from the entry function.
NU_TERMINATED	Terminated.
NU_SLEEP_SUSPEND	Sleeping.
NU_MAILBOX_SUSPEND	Suspended on a mailbox.
NU_QUEUE_SUSPEND	Suspended on a queue.
NU_PIPE_SUSPEND	Suspended on a pipe.
NU_EVENT_SUSPEND	Suspended on an event-flag group.
NU_SEMAPHORE_SUSPEND	Suspended on a semaphore.
NU_MEMORY_SUSPEND	Suspended on a dynamic-memory pool.
NU_PARTITION_SUSPEND	Suspended on a memory-partition pool.
NU_DRIVER_SUSPEND	Suspended from an I/O Driver request.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_TASK	Indicates the task pointer is invalid.

Example

```
NU_TASK          Task;
CHAR             task_name[8];
DATA_ELEMENT     task_status;
UNSIGNED         scheduled_count;
OPTION           priority;
OPTION           preempt;
UNSIGNED         time_slice;
VOID             *stack_base;
UNSIGNED         stack_size;
UNSIGNED         minimum_stack;
STATUS           status;
.
.
.
/* Obtain information about the task control block "Task".
   Assume "Task" has previously been created with the Nucleus
   PLUS NU_Create_Task service call. */
status = NU_Task_Information(&task, task_name, &task_status,
                           &scheduled_count, &priority, &preempt,
                           &time_slice, &stack_base,
                           &stack_size, &minimum_stack);
/* If status is NU_SUCCESS, the other information is accurate. */
```

See Also

NU_Create_Task, NU_Delete_Task, NU_Established_Tasks,
NU_Task_Pointers, NU_Reset_Task



NU_Task_Pointers

```
UNSIGNED NU_Task_Pointers(NU_TASK **pointer_list,
                          UNSIGNED maximum_pointers)
```

This service builds a sequential list of pointers to all established tasks in the system. **Note:** tasks that have been deleted are no longer considered established. The parameter `pointer_list` points to the location for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Control Services

Parameters

Parameter	Meaning
<code>pointer_list</code>	Pointer to an array of NU_TASK pointers. This array will be filled with pointers to established tasks in the system.
<code>maximum_pointers</code>	The maximum number of NU_TASK pointers to place into the array. Typically, this will be the size of the <code>pointer_list</code> array.

Return Value

This service call returns the number of NU_TASK pointers placed into the array.

Example

```
/* Define an array capable of holding 20 task pointers. */
NU_TASK    *Pointer_Array[20];
UNSIGNED    number;

/* Obtain a list of currently active task pointers (Max of 20). */
number = NU_Task_Pointers(&Pointer_Array[0],20);

/* At this point, number contains the actual number of
   pointers in the list. */
```

See Also

NU_Create_Task, NU_Delete_Task, NU_Established_Tasks,
NU_Task_Information, NU_Reset_Task



NU_Terminate_Task

```
STATUS NU_Terminate_Task(NU_TASK *task)
```

This service terminates the task specified by the `task` parameter. **Note 1:** A terminated task cannot execute again until it is reset. **Note 2:** When calling this function from a signal handler, the task whose signal handler is executing cannot be terminated.

Overview

Option	
Tasking Changes	Yes
Allowed From	HISR, Signal Handler, Task
Category	Task Control Services

Parameters

Parameter	Meaning
<code>task</code>	Pointer to the user-supplied task control block. Note: all subsequent requests made to this task require this pointer.

Return Value

Parameter	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_TASK</code>	Indicates the task pointer is invalid.

Example

```
NU_TASK    Task;
STATUS     status;
.
.
.
/* Terminate the task control block "Task".
   Assume "Task" has previously been created with the Nucleus
   PLUS NU_Create_Task service call. */
status = NU_Terminate_Task(&Task);
```

See Also

`NU_Suspend_Task`, `NU_Resume_Task`, `NU_Reset_Task`, `NU_Task_Information`



4

Dynamic Memory

Introduction

Suspension

Dynamic Creation

Determinism

Dynamic Memory Pool Information

Function Reference

Example Source Code



Introduction

A dynamic memory pool contains a user-specified number of bytes. The memory location of the pool is determined by the application. Variable-length allocation and deallocation services are provided for the dynamic memory pool. Allocations are performed in a first-fit manner, i.e. the first available memory that satisfies the request is allocated. If the allocated block is significantly larger than the request, the unused memory is returned to the dynamic memory pool.

Each allocation from a memory pool requires some additional overhead to allow for its pointer structure. This overhead is consumed out of the memory pool from which the allocation is requested. See Section 4.11.2, "Dynamic Memory Data Structures" of the Nucleus PLUS Internals Manual, under the "Dynamic Memory Pool Header Structure" subsection, for full details.

Suspension

The allocate dynamic memory service provides options for unconditional suspension, suspension with a timeout, and no suspension.

A task attempting to allocate dynamic memory from a pool that does not currently have enough available memory may suspend. Resumption of the task is possible when enough previously allocated memory is returned to the pool.

Multiple tasks may suspend on a single dynamic memory pool. Tasks are suspended in either FIFO or priority order, depending on how the dynamic memory pool was created. If the dynamic memory pool supports FIFO suspension, tasks are resumed in the order in which they were suspended. Otherwise, if the dynamic memory pool supports priority suspension, tasks are resumed from high priority to low priority.

Dynamic Creation

Nucleus PLUS dynamic memory pools are created and deleted dynamically. There is no preset limit on the number of dynamic memory pools an application may have. Each dynamic memory pool requires a control block and a pointer to the actual dynamic memory area. The memory for both the control block and the memory area is supplied by the application.

Determinism

Allocating memory from a dynamic memory pool is inherently undeterministic. This is largely due to possible memory fragmentation within the pool. The first-fit algorithm is basically a linear search, and as a result the worst-case performance depends on the amount of fragmentation.

However, memory deallocation is constant. Processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the dynamic memory pool.



Dynamic Memory Pool Information

Application tasks may obtain a list of active dynamic memory pools. Detailed information about each dynamic memory pool is also available. This information includes the dynamic memory pool name, starting pool address, total size, free bytes, number of tasks suspended, and the identity of the first suspended task.

Function Reference

The following function reference contains all functions related to the Nucleus PLUS dynamic memory component. The following functions are contained in this reference:

- NU_Allocate_Memory
- NU_Create_Memory_Pool
- NU_Deallocate_Memory
- NU_Delete_Memory_Pool
- NU_Established_Memory_Pools
- NU_Memory_Pool_Information
- NU_Memory_Pool_Pointers



NU_Allocate_Memory

```
STATUS NU_Allocate_Memory(NU_MEMORY_POOL *pool,
                          VOID **return_pointer,
                          UNSIGNED size,
                          UNSIGNED suspend)
```

This service allocates a block of memory from the specified dynamic memory pool.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Memory Services

Parameters

Parameter	Meaning
pool	Pointer to the dynamic memory pool.
return_pointer	Pointer to the caller's memory pointer. On a successful request, the address of the allocated block is placed in the caller's memory pointer.
size	Specifies the number of bytes to allocate from the dynamic memory pool.
suspend	Specifies whether or not to suspend the calling task if the requested amount of memory is not available.

Suspension

The following table summarizes the possible values for the `suspend` parameter.

Suspension Option	Meaning
NU_NO_SUSPEND	The service returns immediately regardless of whether or not the request can be satisfied. Note: this is the only valid option if the service is called from a non-task thread.
NU_SUSPEND	The calling task is suspended until the requested memory is available.
timeout value	(1 – 4,294,967,293). The calling task is suspended until the memory is available or until the specified number of ticks has expired.



Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_POOL	Indicates the dynamic memory pool is invalid.
NU_INVALID_POINTER	Indicates the return pointer is NULL.
NU_INVALID_SIZE	Indicates an invalid size request.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_NO_MEMORY	Indicates the memory request could not be immediately satisfied.
NU_TIMEOUT	Indicates the requested memory is still unavailable even after suspending for the specified timeout value.
NU_POOL_DELETED	Dynamic memory pool was deleted while the task was suspended.

Example

```

NU_MEMORY_POOL    Pool;
VOID              memory_ptr;
STATUS            status
.
.
.
/* Allocate a 300-byte block of memory with the memory pool
   control block "Pool". If the requested memory is
   unavailable, suspend the calling task unconditionally.
   Assume "Pool" has previously been created with the
   Nucleus PLUS NU_Create_Memory_Pool service call. */
status = NU_Allocate_Memory(&Pool, &memory_ptr, 300, NU_SUSPEND);

/* At this point, status indicates whether the service
   request was successful. */

```

See Also

NU_Deallocate_Memory, NU_Memory_Pool_Information



NU_Create_Memory_Pool

```
STATUS NU_Create_Memory_Pool(NU_MEMORY_POOL *pool, CHAR *name,
                             VOID *start_address,
                             UNSIGNED pool_size,
                             UNSIGNED min_allocation,
                             OPTION suspend_type)
```

This service creates a dynamic memory pool inside a memory area specified by the caller.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Memory Services

Parameters

Parameter	Meaning
pool	Pointer to the user-supplied memory pool control block. Note: all subsequent requests made to the memory pool require this pointer.
name	Pointer to an 8-character name for the memory pool. The name does not have to be null-terminated.
start_address	Specifies the starting address for the memory pool.
pool_size	Specifies the number of bytes in the memory pool.
min_allocation	Specifies the minimum number of bytes in each allocation from this memory pool.
suspend_type	Specifies how tasks suspend on the memory pool. Valid options for this parameter are NU_FIFO and NU_PRIORITY, which represent First-In-First-Out (FIFO) and priority-order task suspension, respectively.



Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_POOL	Indicates the memory pool control block pointer is NULL or is already in use.
NU_INVALID_MEMORY	Indicates the memory area specified by the start address is invalid.
NU_INVALID_SIZE	Indicates the pool size and/or the minimum allocation size is invalid.
NU_INVALID_SUSPEND	Indicates the <code>suspend_type</code> parameter is invalid.

Example

```

/* Assume dynamic memory control block "Pool" is defined as
   a global data structure. This is one of several ways to
   allocate a control block. */

NU_MEMORY_POOL Pool;
.
.
/* Assume status is defined locally. */

STATUS status; /* Memory Pool creation status */

/* Create a dynamic memory pool of 4000-bytes starting
   at the absolute address of 0xA000. Minimum allocation
   size is 30 bytes. Tasks suspend on the pool in order
   of priority. */

status = NU_Create_Memory_Pool(&Pool, "any name", (VOID *) 0xA000,
                               4000, 30, NU_PRIORITY);

/* At this point status indicates if the service was successful. */

```

See Also

```

NU_Delete_Memory_Pool, NU_Established_Memory_Pools,
NU_Memory_Pool_Pointers, NU_Memory_Pool_Information

```



NU_Deallocate_Memory

STATUS NU_Deallocate_Memory(VOID *memory)

This service returns the memory block pointed to by `memory` back to the associated dynamic memory pool.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Memory Services

Parameters

Parameter	Meaning
memory	Pointer to a memory block previously allocated with <code>NU_Allocate_Memory</code> .

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_POINTER	Indicates the memory block pointer is <code>NULL</code> , is not currently allocated, or is invalid.

Example

```
STATUS status;

/* Deallocate the memory block pointed to by "memory." */
status = NU_Deallocate_Memory(memory);

/* At this point status indicates if the service was successful. */
```

See Also

NU_Allocate_Memory, NU_Memory_Pool_Information



NU_Delete_Memory_Pool

```
STATUS NU_Delete_Memory_Pool(NU_MEMORY_POOL *pool)
```

This service deletes a previously created dynamic memory pool. The parameter `pool` identifies the dynamic memory pool to delete. Tasks suspended on this dynamic memory pool are resumed with the appropriate error status. The application must prevent the use of this dynamic memory pool during and after deletion.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Memory Services

Parameters

Parameter	Meaning
<code>pool</code>	Pointer to the user-supplied memory pool control block that has been previously created with <code>NU_Create_Memory_Pool</code> .

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_POOL</code>	Indicates the dynamic memory pool pointer is invalid.

Example

```
NU_MEMORY_POOL    Pool;
STATUS            status;
.
.
/* Delete the memory pool control block "Pool". Assume
   "Pool" has previously been created with the Nucleus
   PLUS NU_Create_Memory_Pool service call. */
status = NU_Delete_Memory_Pool(&Pool);

/* At this point, status indicates whether the service
   request was successful. */
```

See Also

```
NU_Create_Memory_Pool, NU_Established_Memory_Pools,
NU_Memory_Pool_Pointers, NU_Memory_Pool_Information
```



NU_Established_Memory_Pools

UNSIGNED NU_Established_Memory_Pools(VOID)

This service returns the number of established dynamic-memory pools. All created dynamic-memory pools are considered established. Deleted dynamic-memory pools are no longer considered established.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Memory Services

Parameters

None

Return Value

This service call returns the number of created Memory Pools in the system.

Example

```
UNSIGNED total_memory_pools;

/* Obtain the total number of dynamic memory pools. */
total_memory_pools = NU_Established_Memory_Pools( );
```

See Also

NU_Create_Memory_Pool, NU_Delete_Memory_Pool,
NU_Memory_Pool_Pointers, NU_Memory_Pool_Information



NU_Memory_Pool_Information

```
STATUS NU_Memory_Pool_Information(NU_MEMORY_POOL *pool,
                                  CHAR *name,
                                  VOID **start_address,
                                  UNSIGNED *pool_size,
                                  UNSIGNED *min_allocation,
                                  UNSIGNED *available,
                                  OPTION *suspend_type,
                                  UNSIGNED *tasks_waiting,
                                  NU_TASK **first_task)
```

This service returns various information about the specified dynamic memory pool.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Memory Services

Parameters

Parameter	Meaning
pool	Pointer to the dynamic-memory pool.
name	Pointer to an 8 character destination area for the dynamic-memory pool's name.
start_address	Pointer to a memory pointer for holding the starting address of the pool.
pool_size	Pointer to a variable for holding the number of bytes in dynamic memory pool.
min_allocation	Pointer to a variable for holding the minimum number of bytes for each allocation from this pool.
available	Pointer to a variable for holding the number of available bytes in the pool.
suspend_type	Pointer to a variable for holding the task suspend type. Valid task suspend types are <code>NU_FIFO</code> and <code>NU_PRIORITY</code> .
tasks_waiting	Pointer to a variable for holding the number of tasks waiting on the dynamic-memory pool.
first_task	Pointer to a task pointer. The pointer of the first suspended task is placed in this task pointer.

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_POOL</code>	Indicates the dynamic memory pool pointer is invalid.



Example

```
NU_MEMORY_POOL    Pool;
CHAR              pool_name[8];
VOID              *start_address;
UNSIGNED          pool_size;
UNSIGNED          min_allocation;
UNSIGNED          available;
OPTION            suspend_type;
UNSIGNED          tasks_suspended;
NU_TASK           *first_task;
STATUS            status
.
.
.
/* Obtain information about the memory pool control block
   "Pool". Assume "Pool" has previously been created with
   the Nucleus PLUS NU_Create_Memory_Pool service call. */
status = NU_Memory_Pool_Information(&Pool, pool_name,
                                   &start_address, &pool_size,
                                   &min_allocation,
                                   &available, &suspend_type,
                                   &tasks_suspended, &first_task);

/* If status is NU_SUCCESS, the other information is accurate. */
```

See Also

```
NU_Create_Memory_Pool, NU_Delete_Memory_Pool,
NU_Established_Memory_Pools, NU_Memory_Pool_Pointers
```



NU_Memory_Pool_Pointers

```
UNSIGNED NU_Memory_Pool_Pointers(NU_MEMORY_POOL **pointer_list,
                                UNSIGNED maximum_pointers)
```

This service builds a sequential list of pointers to all established dynamic memory pools in the system. **Note:** dynamic-memory pools that have been deleted are no longer considered established. The parameter `pointer_list` points to the location for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Memory Services

Parameters

Parameter	Meaning
<code>pointer_list</code>	Pointer to an array of <code>NU_MEMORY_POOL</code> pointers. This array will be filled with pointers of established memory pools in the system.
<code>maximum_pointers</code>	The maximum number of <code>NU_MEMORY_POOL</code> pointers to place into the array. Typically, this will be the size of the <code>pointer_list</code> array.

Return Value

This service call returns the number of created Memory Pools in the system.

Example

```
/* Define an array capable of holding 20 dynamic memory
   pool pointers. */
NU_MEMORY_POOL *Pointer_Array[20];
UNSIGNED      number;

/* Obtain a list of currently active dynamic-memory
   pool pointers (Maximum of 20). */
number = NU_Memory_Pool_Pointers(&Pointer_Array[0],20);

/* At this point, number contains the actual number of
   pointers in the list. */
```

See Also

```
NU_Create_Memory_Pool, NU_Delete_Memory_Pool,
NU_Established_Memory_Pools, NU_Memory_Pool_Information
```



Example Source Code

The following program demonstrates how the Nucleus PLUS dynamic memory pool component could be used to implement a memory allocation scheme similar to that of the ANSI C malloc and free. A single dynamic memory pool is created out of which all memory requests are allocated. The memory pool is created in the function `memory_init`, and is deleted in `memory_deinit`. All memory can then be allocated through the function calls `memory_allocate`, and `memory_startup_allocate`. The two separate calls are used because, in this example, during a running program we would like for tasks to be suspended when a memory request cannot be immediately satisfied. The function `memory_allocate` could be used during a running program to request memory. When a request cannot be satisfied the calling task would be suspended. However, suspension cannot be requested in the startup function `Application_Initialize`, so a separate function `startup_memory_allocate` is used which does not request suspension when memory requests cannot be immediately satisfied.

Include all necessary Nucleus PLUS include files.

```
#include "nucleus.h"
```

A single `NU_MEMORY_POOL` control block is created. This memory pool control block will be later passed to the `NU_Create_Memory_Pool` service call, which will set up the memory pool for use.

```
NU_MEMORY_POOL System_Memory;
```

In this example, the functions `memory_init`, and `memory_deinit` will be used to initialize and de-initialize the memory pool which is to be used. Specific to Nucleus PLUS, the function `memory_init` will be used to create the memory pool out of which all memory will be allocated. The function `memory_deinit` will be used to delete the dynamic memory pool. Similarly, all memory allocation requests would be made through the `memory_allocate` and `memory_startup_allocate` service calls. Finally, all memory deallocations would be made through the `memory_free` function.

```
VOID *memory_allocate(UNSIGNED alloc_size);
VOID *memory_startup_allocate(UNSIGNED alloc_size);
VOID memory_free(VOID *memory_ptr);
VOID memory_init(VOID *start_addr, UNSIGNED size);
VOID memory_deinit();
```

The function `memory_init` is used to create the dynamic memory pool, `System_Memory`, out of which all memory will be allocated. The function is passed the starting address, and the size of the pool to create. These parameters are then passed to the `NU_Create_Memory_Pool` call to create the memory pool, and associate it with the `System_Memory` control block.



```
VOID memory_init(VOID *start_addr, UNSIGNED size)
{
```

Make the call to `NU_Create_Memory_Pool` to create the dynamic memory pool, and associate the memory pool with the `System_Memory` control block. The `System_Memory` pool will be created such that the minimum allocation request that will be satisfied is a request for 128 bytes of memory. Also, tasks that choose to suspend when a request cannot be satisfied will be resumed in priority order, as indicated by the `NU_PRIORITY` parameter.

```
    if (NU_Create_Memory_Pool(&System_Memory, "sysmem", start_addr,
                             size, 128, NU_PRIORITY) == NU_SUCCESS)
    {
        /* The memory pool was successfully created. */
    }
    else
    {
        /* There was an error creating the memory pool. */
    }
}
```

Use `NU_Delete_Memory_Pool` to delete the memory pool. The only parameter needed by this call is a pointer to the `NU_MEMORY_POOL` control block. Note that any memory allocations that were not deallocated will remain allocated.

```
VOID memory_deinit()
{
    if (NU_Delete_Memory_Pool(&System_Memory) == NU_SUCCESS)
    {
        /* The memory pool was successfully deleted. */
    }
    else
    {
        /* There was an error deleting the memory pool. */
    }
}
```

The function `memory_allocate` would be used to allocate any required memory. The only parameter necessary for this function call is the size, in bytes, of the allocation request. The function will then attempt to allocate the memory with a call to `NU_Allocate_Memory`. If the request is successful (as indicated by the `NU_Allocate_Memory` service call returning `NU_SUCCESS`) then a pointer to the allocated memory is returned to the calling function. Otherwise `NU_NULL` is returned. Note that this is not a reentrant function.

```
VOID *memory_allocate(UNSIGNED alloc_size)
{
```

The void pointer, `temp_ptr` will be used to return the allocated memory to the calling function. It will be passed as a parameter to the `NU_Allocate_Memory` service call. If the call is successful, then `temp_ptr` will contain a valid pointer to the newly allocated memory.



```
VOID *temp_ptr;
```

The `NU_Allocate_Memory` service call will request the memory allocation out of the `System_Memory` dynamic memory pool. If the request can be satisfied, then `temp_ptr` will contain a pointer to the newly allocated memory, and `NU_SUCCESS` will be returned. If the request cannot be immediately satisfied, then the calling task will be suspended, as indicated by the `NU_SUSPEND` parameter. Note that this call should only be used from a task, and not from `Application_Initialize` because suspension cannot be requested from the `Application_Initialize` function.

```
if (NU_Allocate_Memory(&System_Memory, &temp_ptr, alloc_size,
                      NU_SUSPEND) == NU_SUCCESS)
{
    return temp_ptr;
}
else
{
    return NU_NULL;
}
```

Similar to `memory_allocate`, the function `memory_startup_allocate` will use the `NU_Allocate_Memory` service call to request the memory allocation out of the `System_Memory` dynamic memory pool. However, if the request cannot be immediately satisfied, the function `memory_startup_allocate` will not suspend, as indicated by the `NU_NO_SUSPEND` parameter in `NU_Allocate_Memory`. Therefore, this function would be used to allocate memory from the `Application_Initialize` function.

```
VOID *memory_startup_allocate(UNSIGNED alloc_size)
{
    VOID *temp_ptr;
```

Use `NU_Allocate_Memory` to request the allocation out of the `System_Memory` dynamic memory pool.

```
if (NU_Allocate_Memory(&System_Memory, &temp_ptr, alloc_size,
                      NU_NO_SUSPEND) == NU_SUCCESS)
{
    return temp_ptr;
}
else
{
    /* an error occurred allocating memory. */
}
```



The `memory_free` function would be used to deallocate any previously allocated memory. It does this with a call to `NU_Deallocate_Memory`. Note that this function is not reentrant.

```
VOID memory_free(VOID *memory_ptr)
{
```

Use `NU_Deallocate_Memory` to return the memory allocation to the `System_Memory` dynamic memory pool.

```
    if (NU_Deallocate_Memory(&memory_ptr) == NU_SUCCESS)
    {
        /* Memory successfully deallocated. */
    }
    else
    {
        /* An error occurred deallocating memory. */
    }
}
```



5

Partition Memory

Introduction

Function Reference

Example Source Code



Introduction

A partition memory pool contains a specific number of fixed-size memory partitions. The memory location of the pool, the number of bytes in the pool, and the number of bytes in each partition are determined by the application. Individual partitions are allocated and deallocated from the partition memory pool.

Allocation from a memory pool requires some additional overhead to allow for its pointer structure. See Section 4.10.2, "Partition Memory Data Structures" of the Nucleus PLUS Internals Manual, under the "Partition Memory Pool Header Structure" subsection, for full details.

Suspension

The allocate partition service provides options for unconditional suspension, suspension with a timeout, and no suspension.

A task attempting to allocate a partition from an empty pool can suspend. Resumption of that task is possible when a partition is returned to the pool.

Multiple tasks may suspend on a single partition memory pool. Tasks are suspended in either FIFO or priority order, depending on how the partition memory pool was created. If the partition memory pool supports FIFO suspension, tasks are resumed in the order in which they were suspended. Otherwise, if the partition memory pool supports priority suspension, tasks are resumed from high priority to low priority.

Dynamic Creation

Nucleus PLUS partition memory pools are created and deleted dynamically. There is no preset limit on the number of partition memory pools an application may have. Each partition memory pool requires a control block and a pointer to the memory area for the partition. The memory for both the control block and the partition area is supplied by the application.

Determinism

Since searching is completely avoided, processing required for allocating and deallocating partitions is fast and constant. However, the processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the partition memory pool.



Partition Information

Application tasks may obtain a list of active partition memory pools. Detailed information about each partition memory pool is also available. This information includes the partition memory pool name, starting pool address, total partitions, partition size, remaining partitions, number of tasks suspended, and the identity of the first suspended task.

Function Reference

The following function reference contains all functions related to the Nucleus PLUS partition memory component. The following functions are contained in this reference:

```
NU_Allocate_Partition  
NU_Create_Partition_Pool  
NU_Deallocate_Partition  
NU_Delete_Partition_Pool  
NU_Established_Partition_Pools  
NU_Partition_Pool_Information  
NU_Partition_Pool_Pointers
```



NU_Allocate_Partition

```
STATUS NU_Allocate_Partition(NU_PARTITION_POOL *pool,
                             VOID **return_pointer,
                             UNSIGNED suspend)
```

This service allocates a memory partition from the specified memory partition pool. Note that the size of the memory partition is defined when the memory partition pool is created.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Memory Services

Parameters

Parameter	Meaning
pool	Pointer to the memory partition pool.
return_pointer	Pointer to the caller's memory pointer. On a successful request, the address of the allocated memory partition is placed in the caller's memory pointer.
suspend	Specifies whether or not to suspend the calling task if there are no memory partitions available.

Suspension

The following table summarizes the possible values for the `suspend` parameter.

Suspension Option	Meaning
NU_NO_SUSPEND	The service returns immediately regardless of whether or not the request can be satisfied. Note: this is the only valid option if the service is called from a non-task thread.
NU_SUSPEND	The calling task is suspended until a memory partition is available.
timeout value	(1 - 4,294,967,293). The calling task is suspended until a memory partition is available, or until the specified number of ticks has expired.



Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_POOL	Indicates the memory partition pool pointer is invalid.
NU_INVALID_POINTER	Indicates the return pointer is NULL.
NU_INVALID_SUSPEND	Indicates that a suspend was attempted from a non-task thread.
NU_NO_PARTITION	Indicates the memory partition request could not be immediately satisfied.
NU_TIMEOUT	Indicates that no memory partition is available even after suspending for the specified timeout value.
NU_POOL_DELETED	Partition memory pool was deleted while the task was suspended.

Example

```

NU_PARTITION_POOL Pool;
VOID                *memory_ptr;
STATUS              status

/* Allocate a memory partition with the memory partition
   pool control block "Pool". If there are no partitions
   available, suspend the calling task unconditionally.
   Assume "Pool" has previously been created with the
   Nucleus PLUS NU_Create_Partition_Pool service call.*/

status = NU_Allocate_Partition(&Pool, &memory_ptr,
                              NU_SUSPEND);

/* At this point, status indicates whether the
   service request was successful. */

```

See Also

```

NU_Create_Partition_Pool, NU_Deallocate_Partition,
NU_Partition_Pool_Information

```



NU_Create_Partition_Pool

```
STATUS NU_Create_Partition_Pool(NU_PARTITION_POOL *pool,
                                CHAR *name, VOID *start_address,
                                UNSIGNED pool_size,
                                UNSIGNED partition_size,
                                OPTION suspend_type)
```

This service creates a pool of fixed-size memory partitions inside a memory area specified by the caller.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Memory Services

Parameters

Parameter	Meaning
pool	Pointer to the user-supplied partition pool control block. Note: all subsequent requests made to this partition pool require this pointer.
name	Pointer to an 8-character name for the partition pool. The name does not have to be null-terminated.
start_address	Specifies the starting address for the fixed-size memory partition pool.
pool_size	Specifies the total number of bytes in the memory area.
partition_size	Specifies the number of bytes for each partition in the pool. There is a small amount of memory "overhead" associated with each partition. This overhead is required by the two data pointers used.
suspend_type	Specifies how tasks suspend on the partition pool. Valid options for this parameter are <code>NU_FIFO</code> and <code>NU_PRIORITY</code> , which represent First-In-First-Out (FIFO) and priority-order task suspension, respectively.



Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_POOL	Indicates the partition pool control block pointer is NULL or is already in use.
NU_INVALID_MEMORY	Indicates the memory area specified by the <code>start_address</code> is invalid.
NU_INVALID_SIZE	Indicates the partition size is either 0 or larger than the total partition memory area.
NU_INVALID_SUSPEND	Indicates the <code>suspend_type</code> parameter is invalid.

Example

```

/* Assume partition memory control block "Pool" is defined
   as a global data structure. This is one of several ways
   to allocate a control block. */

NU_PARTITION_POOL Pool;
.
.
/* Assume status is defined locally. */

STATUS status; /* Partition Pool creation status */

/* Create a partition memory pool of 40-byte memory partitions,
   in a 2000-byte memory area starting at the absolute address
   of 0xB000. Task suspend on the pool in FIFO order. */
status = NU_Create_Partition_Pool(&Pool, "any name",
                                   (VOID *) 0xB000, 2000,
                                   40, NU_FIFO);

/* At this point status indicates if the service was
   successful. */

```

See Also

```

NU_Delete_Partition_Pool, NU_Established_Partition_Pools,
NU_Partition_Pool_Pointers, NU_Partition_Pool_Information

```



NU_Deallocate_Partition

```
STATUS NU_Deallocate_Partition(VOID *partition)
```

This service returns the memory partition pointed to by `partition` back to the associated pool.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Memory Services

Parameters

Parameter	Meaning
memory	Pointer to a memory partition previously allocated with <code>NU_Allocate_Partition</code> .

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_POINTER	Indicates the memory partition pointer is <code>NULL</code> , is not currently allocated, or is invalid.

Example

```
STATUS      status;

/* Deallocate the memory partition pointed to by "partition." */
status = NU_Deallocate_Partition(partition);

/* At this point status indicates if the service was
   successful. */
```

See Also

`NU_Allocate_Partition`, `NU_Partition_Pool_Information`



NU_Delete_Partition_Pool

```
STATUS NU_Delete_Partition_Pool(NU_PARTITION_POOL *pool)
```

This service deletes a previously created memory partition pool. The parameter `pool` identifies the memory partition pool to delete. Tasks suspended on this memory partition pool are resumed with the appropriate error status. The application must prevent the use of this memory partition pool during and after deletion.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Memory Services

Parameters

Parameter	Meaning
<code>pool</code>	Pointer to the user-supplied partition pool control block that has been previously created with <code>NU_Create_Partition_Pool</code> .

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_POOL</code>	Indicates the memory partition pool pointer is invalid.

Example

```
NU_PARTITION_POOL Pool;
STATUS          status
.
.
/* Delete the partition pool control block "Pool".
   Assume "Pool" has previously been created with the
   Nucleus PLUS NU_Create_Partition_Pool service call.*/
status = NU_Delete_Partition_Pool(&Pool);

/* At this point, status indicates whether the service
   request was successful. */
```

See Also

```
NU_Create_Partition_Pool, NU_Established_Partition_Pools,
NU_Partition_Pool_Pointers, NU_Partition_Pool_Information
```



NU_Established_Partition_Pools

UNSIGNED NU_Established_Partition_Pools(VOID)

This service returns the number of established memory-partition pools. All created memory-partition pools are considered established. Deleted memory-partition pools are no longer considered established.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Memory Services

Parameters

None

Return Value

This service call returns the number of created partition pools in the system.

Example

```
UNSIGNED total_partition_pools;

/* Obtain the total number of memory partition pools. */
total_partition_pools = NU_Established_Partition_Pools();
```

See Also

NU_Create_Partition_Pool, NU_Delete_Partition_Pool,
NU_Partition_Pool_Pointers, NU_Partition_Pool_Information



NU_Partition_Pool_Information

```
STATUS NU_Partition_Pool_Information(NU_PARTITION_POOL *pool,
                                     CHAR *name,
                                     VOID **start_address,
                                     UNSIGNED *pool_size,
                                     UNSIGNED *partition_size,
                                     UNSIGNED *available,
                                     UNSIGNED *allocated,
                                     OPTION *suspend_type,
                                     UNSIGNED *tasks_waiting,
                                     NU_TASK **first_task)
```

This service returns various information about the specified partition memory pool.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, task
Category	Memory Services

Parameters

Parameter	Meaning
pool	Pointer to the partition pool.
name	Pointer to an 8 character destination area for the partition pool's name.
start_address	Pointer to a memory pointer for holding the starting address of the pool.
pool_size	Pointer to a variable for holding the total number of bytes in the partition pool.
partition_size	Pointer to a variable for holding the number of bytes in each memory partition.
available	Pointer to a variable for holding the number of available partitions in the pool.
allocated	Pointer to a variable for holding the number of allocated pool partitions.
suspend_type	Pointer to a variable for holding the task suspend type. Valid task suspend types are NU_FIFO and NU_PRIORITY.
tasks_waiting	Pointer to a variable for holding the number of tasks waiting on the partition pool.
first_task	Pointer to a task pointer. The pointer of the first suspended task is placed in this task pointer.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_POOL	Indicates the partition pool pointer is invalid.

Example

```

NU_PARTITION_POOL Pool;
CHAR                pool_name[8];
VOID                *start_address;
UNSIGNED            pool_size;
UNSIGNED            partition_size;
UNSIGNED            available;
UNSIGNED            allocated;
OPTION              suspend_type;
UNSIGNED            tasks_suspended;
NU_TASK             *first_task;
STATUS              status
.
.
/* Obtain information about the partition pool control
block "Pool". Assume "Pool" has previously been created
with the Nucleus PLUS NU_Create_Partition_Pool service call. */
status = NU_Partition_Pool_Information(&Pool, pool_name,
                                     &start_address, &pool_size,
                                     &partition_size, &available,
                                     &allocated, &suspend_type,
                                     &tasks_suspended,
                                     &first_task);

/* If status is NU_SUCCESS, the other information is accurate. */

```

See Also

```

NU_Create_Partition_Pool, NU_Delete_Partition_Pool,
NU_Established_Partition_Pools, NU_Partition_Pool_Pointers

```



NU_Partition_Pool_Pointers

```
UNSIGNED NU_Partition_Pool_Pointers(NU_PARTITION_POOL **pointer_list,
                                     UNSIGNED maximum_pointers)
```

This service builds a sequential list of pointers to all established memory partition pools in the system. **Note:** memory partition pools that have been deleted are no longer considered established. The parameter `pointer_list` points to the location used for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Memory Services

Parameters

Parameter	Meaning
<code>pointer_list</code>	Pointer to an array of NU_PARTITION_POOL pointers. This array will be filled with pointers of established partition pools in the system.
<code>maximum_pointers</code>	The maximum number of NU_PARTITION_POOL pointers to place into the array. Typically, this will be the size of the <code>pointer_list</code> array.

Return Value

This service call returns the number of created Memory Pools in the system.

Example

```
/* Define an array capable of holding 20 memory
   partition pool pointers. */
NU_PARTITION_POOL *Pointer_Array[20];
UNSIGNED          number;

/* Obtain a list of currently active memory partition
   pool pointers (Maximum of 20). */
number = NU_Partition_Pool_Pointers(&Pointer_Array[0], 20);

/* number contains the actual number of pointers in the list. */
```

See Also

```
NU_Create_Partition_Pool, NU_Delete_Partition_Pool,
NU_Established_Partition_Pools, NU_Partition_Pool_Information
```



Example Source Code

The following program demonstrates how the Nucleus PLUS partition memory pool component could be used to implement a memory allocation scheme similar to that of the ANSI C malloc and free. A single partition memory pool is created out of which all memory requests are allocated. The memory pool is created in the function `memory_init`, and is deleted in `memory_deinit`. All memory can then be allocated through the function calls `memory_allocate`, and `memory_startup_allocate`. The two separate calls are used because, in this example, during a running program we would like for tasks to be suspended when a memory request cannot be immediately satisfied. The function `memory_allocate` could be used during a running program to request memory. When a request cannot be satisfied the calling task would be suspended. However, suspension cannot be requested in the startup function `Application_Initialize`, so a separate function `startup_memory_allocate` is used which does not request suspension when memory requests cannot be immediately satisfied.

Include all necessary Nucleus PLUS include files.

```
#include "nucleus.h"
```

A single `NU_PARTITION_POOL` control block is created. This partition pool control block will be later passed to the `NU_Create_Partition_Pool` service call, which will set up the partition pool for use.

```
NU_SEMAPHORE semaphore_memory;  
NU_PARTITION_POOL System_Memory;
```

In this example, the functions `memory_init`, and `memory_deinit` will be used to initialize and de-initialize the partition memory pool which is to be used. Specific to Nucleus PLUS, the function `memory_init` will be used to create the partition pool out of which all memory will be allocated. The function `memory_deinit` will be used to delete the partition memory pool. Similarly, all memory allocation requests would be made through the `memory_allocate` and `memory_startup_allocate` service calls. Finally, all memory deallocations would be made through the `memory_free` function.

```
VOID *memory_allocate();  
VOID *memory_startup_allocate();  
VOID memory_free(VOID *memory_ptr);  
VOID memory_init(VOID *start_addr, UNSIGNED size, UNSIGNED  
                partition_size);  
VOID memory_deinit();
```

The function `memory_init` is used to create the partition memory pool, `System_Memory`, out of which all memory will be allocated. The function is passed the starting address, the size of the pool to create, and the size of each partition to be allocated. These parameters are then passed to the `NU_Create_Partition_Pool` call to create the memory pool, and associate it with the `System_Memory` control block.




```

VOID memory_init(VOID *start_addr, UNSIGNED size, UNSIGNED
                 partition_size)
{

```

Make the call to `NU_Create_Partition_Pool` to create the partition pool, and associate the memory pool with the `System_Memory` control block. As previously mentioned, the `System_Memory` partition pool will be created with the starting address, size, and partition size as specified in the function parameters. The partition pool will also be created such that tasks which choose to suspend when a request cannot be satisfied will be resumed in priority order, as indicated by the `NU_PRIORITY` parameter.

```

    if (NU_Create_Partition_Pool(&System_Memory, "sysmem", start_addr,
                                size, partition_size, NU_PRIORITY)
        == NU_SUCCESS)
    {
        /* Partition pool successfully created. */
    }
    else
    {
        /* Error creating partition pool. */
    }
}

```

Use `NU_Delete_Partition_Pool` to delete the memory pool. The only parameter needed by this call is a pointer to the `NU_PARTITION_POOL` control block. Note that any memory allocations that were not deallocated will remain allocated.

```

VOID memory_deinit()
{
    if (NU_Delete_Partition_Pool(&System_Memory) == NU_SUCCESS)
    {
        /* Partition pool successfully deleted. */
    }
    else
    {
        /* Error deleting partition pool. */
    }
}

```

The function `memory_allocate` would be used to allocate any required memory. Note that this function does not take any parameters, unlike its dynamic memory counterpart. Since all allocations are made in the size that was specified when the pool was created, the size parameter is not necessary.

The function will attempt to allocate the memory with a call to `NU_Allocate_Memory`. If the request is successful (as indicated by the `NU_Allocate_Memory` service call returning `NU_SUCCESS`) then a pointer to the allocated memory is returned to the calling function. Otherwise `NU_NULL` is returned. Note that is not a reentrant function.

```

VOID *memory_allocate()
{

```

The void pointer, `temp_ptr` will be used to return the allocated memory to the calling function. It will be passed as a parameter to the `NU_Allocate_Partition` service call. If the call is successful, then `temp_ptr` will contain a valid pointer to the newly allocated memory.

```
VOID *temp_ptr;
```

The `NU_Allocate_Partition` service call will request the memory allocation out of the `System_Memory` partition memory pool. If the request can be satisfied, then `temp_ptr` will contain a pointer to the newly allocated memory, and `NU_SUCCESS` will be returned. If the request cannot be immediately satisfied, then the calling task will be suspended, as indicated by the `NU_SUSPEND` parameter. Note that this call should only be used from a task, and not from the `Application_Initialize` because suspension cannot be requested from the `Application_Initialize` function.

```
if (NU_Allocate_Partition(&System_Memory, &temp_ptr, NU_SUSPEND)
    == NU_SUCCESS)
{
    return temp_ptr;
}
else
{
}
}
```

Similar to `memory_allocate`, the function `memory_startup_allocate` will use the `NU_Allocate_Partition` service call to request the memory allocation out of the `System_Memory` partition memory pool. However, if the request cannot be immediately satisfied, the function `memory_startup_allocate` will not suspend, as indicated by the `NU_NO_SUSPEND` parameter in `NU_Allocate_Memory`. Therefore, this function would be used to allocate memory from the `Application_Initialize` function.

```
VOID *memory_startup_allocate()
{
    VOID *temp_ptr;
```

Use `NU_Allocate_Partition` to request the allocation out of the `System_Memory` partition memory pool.

```
if (NU_Allocate_Partition(&System_Memory, &temp_ptr, NU_NO_SUSPEND)
    == NU_SUCCESS)
{
    return temp_ptr;
}
else
{
    /* Error in memory allocation. */
}
```



The `memory_free` function would be used to deallocate any previously allocated memory. It does this with a single call to `NU_Deallocate_Partition`. Note that this function is not reentrant.

```
VOID memory_free(VOID *memory_ptr)
{
```

Use `NU_Deallocate_Memory` to return the memory allocation to the `System_Memory` partition memory pool.

```
    if (NU_Deallocate_Partition(&memory_ptr) == NU_SUCCESS)
    {
    }
    else
    {
    }
}
```



6

Mailboxes

Introduction

Function Reference

Example Source Code



Introduction

Mailboxes provide a low-overhead mechanism to transmit simple messages. Each mailbox is capable of holding a single message the size of four 32-bit words. Messages are sent and received by value. A send message request copies the message into the mailbox, while a receive message request copies the message out of the mailbox.

Suspension

Send and receive mailbox services provide options for unconditional suspension, suspension with a timeout, and no suspension.

Tasks can suspend on a mailbox for several reasons. A task attempting to receive a message from an empty mailbox can suspend. Also, a task attempting to send a message to a non-empty mailbox can suspend. A suspended task is resumed when the mailbox is able to satisfy that task's request. For example, suppose a task is suspended on a mailbox waiting to receive a message. When a message is sent to the mailbox, the suspended task is resumed.

Multiple tasks can suspend on a single mailbox. Tasks are suspended in either FIFO or priority order, depending on how the mailbox was created. If the mailbox supports FIFO suspension, tasks are resumed in the order in which they were suspended. Otherwise, if the mailbox supports priority suspension, tasks are resumed from high priority to low priority.

Broadcast

A mailbox message may be broadcast. This service is similar to a send request, except that all tasks waiting for a message from the mailbox are given the broadcast message.

Dynamic Creation

Nucleus PLUS mailboxes are created and deleted dynamically. There is no preset limit on the number of mailboxes an application may have. Each mailbox requires a control block. The memory for the control block is supplied by the application.

Determinism

Processing time required for sending and receiving mailbox messages is constant. However, the processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the mailbox.



Mailbox Information

Application tasks may obtain a list of active mailboxes. Detailed information about each mailbox can also be obtained. This information includes the mailbox name, suspension type, whether a message is present, and the first task waiting.

Function Reference

The following function reference contains all functions related to Nucleus PLUS mailboxes. The following functions are contained in this reference:

- NU_Broadcast_To_Mailbox
- NU_Create_Mailbox
- NU_Delete_Mailbox
- NU_Established_Mailboxes
- NU_Mailbox_Information
- NU_Mailbox_Pointers
- NU_Receive_From_Mailbox
- NU_Reset_Mailbox
- NU_Send_To_Mailbox



NU_Broadcast_To_Mailbox

```
STATUS NU_Broadcast_To_Mailbox(NU_MAILBOX *mailbox,
                               VOID *message,
                               UNSIGNED suspend)
```

This service broadcasts a message to all tasks waiting for a message from the specified mailbox. If no tasks are waiting, the message is simply placed in the mailbox. Each message is equivalent in size to four `UNSIGNED` data elements.

Overview

Option	
Tasking Changes	Yes
Allowed From	<code>Application_Initialize</code> , HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
<code>mailbox</code>	Pointer to the mailbox.
<code>message</code>	Pointer to the broadcast message.
<code>suspend</code>	Specifies whether or not to suspend the calling task if the mailbox already contains a message.

Suspension

The following table summarizes the possible values for the `suspend` parameter.

Suspension Option	Meaning
<code>NU_NO_SUSPEND</code>	The service returns immediately regardless of whether or not the request can be satisfied. Note: this is the only valid option if the service is called from a non-task thread.
<code>NU_SUSPEND</code>	The calling task is suspended until the message can be copied into the mailbox.
<code>timeout value</code>	(1 – 4,294,967,293). The calling task is suspended until the message can be copied into the mailbox or until the specified number of ticks has expired.



Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_MAILBOX	Indicates the mailbox pointer is invalid.
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_MAILBOX_FULL	Indicates the message could not be immediately placed in the mailbox because the mailbox already contains a message.
NU_TIMEOUT	Indicates that the mailbox is still unable to accept the message even after suspending for the specified timeout value.
NU_MAILBOX_DELETED	Mailbox was deleted while the task was suspended.
NU_MAILBOX_RESET	Mailbox was reset while the task was suspended.

Example

```

NU_MAILBOX Mailbox;
UNSIGNED   message[4];
STATUS     status
.
.
.
/* Build a message to send to a mailbox. The
   contents of "message" are not significant */

message[0] = 0x00001111;
message[1] = 0x22223333;
message[2] = 0x44445555;
message[3] = 0x66667777;

/* Send the message to the mailbox control block "Mailbox". If the
   mailbox already contains a message, suspend for 20 timer ticks.
   Assume "Mailbox" has previously been created with the Nucleus
   PLUS NU_Create_Mailbox service call. */
status = NU_Broadcast_To_Mailbox(&Mailbox, &message[0], 20);

/* At this point, status indicates whether the
   service request was successful. */

```

See Also

NU_Send_To_Mailbox, NU_Receive_From_Mailbox, NU_Mailbox_Information



NU_Create_Mailbox

```
STATUS NU_Create_Mailbox(NU_MAILBOX *mailbox,
                        CHAR *name,
                        OPTION suspend_type)
```

This service creates a task communication mailbox. A mailbox is capable of holding a single message. Mailbox messages are equivalent in size to four UNSIGNED data elements.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
mailbox	Pointer to the user-supplied mailbox control block. Note: all subsequent requests made to the mailbox require this pointer.
name	Pointer to an 8 character name for the mailbox. The name does not have to be null-terminated.
suspend_type	Specifies how tasks suspend on the mailbox. Valid options for this parameter are NU_FIFO and NU_PRIORITY, which represent First-In-First-Out (FIFO) and priority-order task suspension, respectively.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_MAILBOX	Indicates the mailbox control block pointer is NULL or is already in use.
NU_INVALID_SUSPEND	Indicates the suspend_type parameter is invalid.



Example

```

/* Assume mailbox control block "Mailbox" is defined as a global
   data structure. This is one of several ways to allocate a
   control block. */
   NU_MAILBOX Mailbox;
   .
   .
/* Assume status is defined locally. */

   STATUS      status; /* Mailbox creation status */

/* Create a mailbox that manages task suspension in a FIFO manner. */
   status =  NU_Create_Mailbox(&Mailbox, "any name", NU_FIFO);

/* At this point status indicates if the service was successful. */

```

See Also

NU_Delete_Mailbox, NU_Established_Mailboxes, NU_Mailbox_Pointers,
 NU_Mailbox_Information



NU_Delete_Mailbox

```
STATUS NU_Delete_Mailbox(NU_MAILBOX *mailbox)
```

This service deletes a previously created mailbox. The parameter `mailbox` identifies the mailbox to delete. Tasks suspended on this mailbox are resumed with the appropriate error status. The application must prevent the use of this mailbox during and after deletion.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
<code>mailbox</code>	Pointer to the user-supplied mailbox control block.

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_MAILBOX</code>	Indicates the mailbox pointer is invalid.

Example

```
NU_MAILBOX Mailbox;
STATUS      status
.
.
.
/* Delete the mailbox control block "Mailbox". Assume "Mailbox"
   has previously been created with the Nucleus PLUS
   NU_Create_Mailbox service call. */
status = NU_Delete_Mailbox(&Mailbox);

/* At this point, status indicates whether the
   service request was successful. */
```

See Also

```
NU_Create_Mailbox, NU_Established_Mailboxes, NU_Mailbox_Pointers,
NU_Mailbox_Information
```



NU_Established_Mailboxes

UNSIGNED NU_Established_Mailboxes(VOID)

This service returns the number of established mailboxes. All created mailboxes are considered established. Deleted mailboxes are no longer considered established.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

None

Return Value

This service call returns the number of created mailboxes in the system.

Example

```
UNSIGNED total_mailboxes;

/* Obtain the total number of mailboxes. */
total_mailboxes = NU_Established_Mailboxes( );
```

See Also

NU_Create_Mailbox, NU_Delete_Mailbox, NU_Mailbox_Pointers,
NU_Mailbox_Information

NU_Mailbox_Information

```
STATUS NU_Mailbox_Information(NU_MAILBOX *mailbox,
                             CHAR *name,
                             OPTION *suspend_type,
                             DATA_ELEMENT *message_present,
                             UNSIGNED *tasks_waiting,
                             NU_TASK **first_task)
```

This service returns various information about the specified mailbox.

Overview

Option	
Tasking Changes	No
Allowed From	Application Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
mailbox	Pointer to the user-supplied mailbox control block.
name	Pointer to an 8 character destination area for the mailbox's name.
suspend_type	Pointer to a variable for holding the task suspend type. Valid task suspend types are NU_FIFO and NU_PRIORITY.
message_present	If a message is present in the mailbox, an NU_TRUE value is placed in the variable pointed to by this parameter. Otherwise, if the mailbox is empty, an NU_FALSE value is placed in the variable.
tasks_waiting	Pointer to a variable for holding the number of tasks waiting on the mailbox.
first_task	Pointer to a task pointer. The pointer of the first suspended task is place in the task pointer.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_MAILBOX	Indicates the mailbox pointer is invalid.



Example

```

NU_MAILBOX      Mailbox;
CHAR            mailbox_name[8];
OPTION          suspend_type;
DATA_ELEMENT    message_present;
UNSIGNED        tasks_suspended;
NU_TASK         *first_task;
STATUS          status
.
.
.
/* Obtain information about the mailbox control block "Mailbox".
   Assume "Mailbox" has previously been created with the Nucleus
   PLUS NU_Create_Mailbox service call. */
status = NU_Mailbox_Information(&Mailbox, mailbox_name,
                               &suspend_type, &message_present,
                               &tasks_suspended, &first_task);

/* If status is NU_SUCCESS, the other information is accurate. */

```

See Also

NU_Create_Mailbox, NU_Delete_Mailbox, NU_Established_Mailboxes,
NU_Mailbox_Pointers



NU_Mailbox_Pointers

```
UNSIGNED NU_Mailbox_Pointers(NU_MAILBOX **pointer_list,
                             UNSIGNED maximum_pointers)
```

This service builds a sequential list of pointers to all established mailboxes in the system.

Note: mailboxes that have been deleted are no longer considered established. The parameter `pointer_list` points to the location used for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
<code>pointer_list</code>	Pointer to an array of <code>NU_MAILBOX</code> pointers. This array will be filled with pointers of established mailboxes in the system.
<code>maximum_pointers</code>	The maximum number of <code>NU_MAILBOX</code> pointers to place into the array. Typically this will be the size of the <code>pointer_list</code> array.

Return Value

This service call returns the number of created mailboxes in the system.



Example

```
/* Define an array capable of holding 20 mailbox pointers */
NU_MAILBOX *Pointer_Array[20];
UNSIGNED   number;

/* Obtain a list of currently active mailbox
   pointers (Maximum of 20). */
number = NU_Mailbox_Pointers(&Pointer_Array[0], 20);

/* At this point, the number contains the actual number of
   pointers in the list. */
```

See Also

NU_Create_Mailbox, NU_Delete_Mailbox, NU_Established_Mailboxes,
NU_Mailbox_Information



NU_Receive_From_Mailbox

```
STATUS NU_Receive_From_Mailbox(NU_MAILBOX *mailbox,
                               VOID *message,
                               UNSIGNED suspend)
```

This service retrieves a message from the specified mailbox. If the mailbox contains a message, it is immediately removed from the mailbox and copied into the designated location. Mailbox messages are equivalent in size to four `UNSIGNED` data elements.

Overview

Option	
Tasking Changes	Yes
Allowed From	<code>Application_Initialize</code> , <code>HISR</code> , <code>Signal Handler</code> , <code>Task</code>
Category	Task Communication Services

Parameters

Parameter	Meaning
<code>mailbox</code>	Pointer to the user-supplied mailbox control block.
<code>message</code>	Pointer to the message destination. Note: message destination must be at least the size of four <code>UNSIGNED</code> data elements.
<code>suspend</code>	Specifies whether or not to suspend the calling task if the mailbox is empty.

Suspension

The following table summarizes the possible values for the `suspend` parameter.

Suspension Option	Meaning
<code>NU_NO_SUSPEND</code>	The service returns immediately regardless of whether or not the request can be satisfied. Note: this is the only valid option if the service is called from a non-task thread.
<code>NU_SUSPEND</code>	The calling task is suspended until a message is available.
<code>timeout value</code>	(1 – 4,294,967,293). The calling task is suspended until a message is available or until the specified number of ticks has expired.



Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_MAILBOX	Indicates the mailbox pointer is invalid.
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_MAILBOX_EMPTY	Indicates the mailbox is empty.
NU_TIMEOUT	Indicates that the mailbox is still empty even after suspending for the specified timeout value.
NU_MAILBOX_DELETED	Mailbox was deleted while the task was suspended.
NU_MAILBOX_RESET	Mailbox was reset while the task was suspended.

Example

```

NU_MAILBOX mailbox;
UNSIGNED message[4];
STATUS status;
.
.
.
/* Receive a message from the mailbox control block "Mailbox".
   If the mailbox is empty, suspend for 20 timer ticks. Note:
   the order of multiple tasks suspending on the same mailbox
   is determined when the mailbox is created. Assume "Mailbox"
   has previously been created with the Nucleus PLUS
   NU_Create_Mailbox service call. */
status = NU_Receive_From_Mailbox(&Mailbox,&message[0],20);

/* At this point, status indicates whether the service request
   was successful. If successful, "message" contains the
   received mailbox message. */

```

See Also

NU_Broadcast_To_Mailbox, NU_Send_To_Mailbox, NU_Mailbox_Information



NU_Reset_Mailbox

```
STATUS NU_Reset_Mailbox(NU_MAILBOX *mailbox)
```

This service discards a message currently in the mailbox specified by `mailbox`. All tasks suspended on the mailbox are resumed with the appropriate reset status.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
<code>mailbox</code>	Pointer to the user-supplied mailbox control block.

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_MAILBOX</code>	Indicates the mailbox pointer is invalid.

Example

```
NU_MAILBOX Mailbox;
STATUS      status;
.
.
.
/* Reset the mailbox control block "Mailbox".
   Assume "Mailbox" has previously been created with
   the Nucleus PLUS NU_Create_Mailbox service call. */
status = NU_Reset_Mailbox(&Mailbox);
```

See Also

`NU_Broadcast_To_Mailbox`, `NU_Send_To_Mailbox`,
`NU_Receive_From_Mailbox`, `NU_Mailbox_Information`



NU_Send_To_Mailbox

```
STATUS NU_Send_To_Mailbox(NU_MAILBOX *mailbox,
                        VOID *message,
                        UNSIGNED suspend)
```

This service places a message into the specified mailbox. If the mailbox is empty, the message is copied immediately into the mailbox. Mailbox messages are equivalent to four UNSIGNED data elements in size. The parameters of this service are further defined as follows:

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, task
Category	Task Communication Services

Parameters

Parameter	Meaning
mailbox	Pointer to the mailbox.
message	Pointer to the message to send.
suspend	Specifies whether or not to suspend the calling task if the mailbox already contains a message.

Suspension

The following table summarizes the possible values for the suspend parameter.

Suspension Option	Meaning
NU_NO_SUSPEND	The service returns immediately regardless of whether or not the request can be satisfied. Note: this is the only valid option if the service is called from a non-task thread.
NU_SUSPEND	The calling task is suspended until the message can be sent.
timeout value	(1 - 4,294,967,293). The calling task is suspended until the message can be sent or until the specified number of ticks has expired.



Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_MAILBOX	Indicates the mailbox pointer is invalid.
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_MAILBOX_FULL	Indicates the mailbox is full.
NU_TIMEOUT	Indicates that the mailbox is still full even after suspending for the specified timeout value.
NU_MAILBOX_DELETED	Mailbox was deleted while the task was suspended.
NU_MAILBOX_RESET	Mailbox was reset while the task was suspended.

Example

```

NU_MAILBOX Mailbox;
UNSIGNED  message[4];
STATUS    status;
.
.
.
/* Build a 4 UNSIGNED-variable message to send.
   The contents of "message" have no significance. */
message[0] = 0x00001111;
message[1] = 0x00002222;
message[2] = 0x00003333;
message[3] = 0x00004444;

/* Send the message to the mailbox control block
   "Mailbox". Suspend the calling task until the
   message can be sent or until 25 timer ticks expire.
   Assume "Mailbox" has previously been created with
   the Nucleus PLUS NU_Create_Mailbox service call. */
status = NU_Send_To_Mailbox(&Mailbox, &message[0], 25);

/* At this point, status indicates whether the service
   request was successful. If successful, "message" was
   sent to "Mailbox". */

```

See Also

```

NU_Broadcast_To_Mailbox, NU_Receive_From_Mailbox,
NU_Mailbox_Information

```



Example Source Code

The following example will demonstrate the use of Nucleus PLUS mailboxes to communicate between tasks.

Include all necessary Nucleus PLUS include files.

```
#include "nucleus.h"
```

Create structures for three tasks (NU_TASK), and the memory pool (NU_MEMORY_POOL) out of which all memory will be allocated for task stacks. Also create a mailbox structure (NU_MAILBOX). This mailbox will be used to communicate between the three tasks in the system.

```
NU_TASK task_rcv_1;
NU_TASK task_rcv_2;
NU_TASK task_send;
NU_MAILBOX mailbox_comm;
NU_MEMORY_POOL dm_memory;
```

Three void pointers will be used in this example. Each void pointer will hold a pointer to a separate task stack. Although not demonstrated in this program, these pointers could be used at a later time in the program to deallocate the task stacks, or they could be discarded if the task stacks will never be deallocated.

```
VOID *stack_rcv_1;
VOID *stack_rcv_2;
VOID *stack_send;
```

Declare the task entry point function for each of the three tasks. These will later be passed as a parameter to the NU_Create_Task call which will associate these functions with each of their respective tasks.

```
void entry_rcv_1(UNSIGNED argc, VOID *argv);
void entry_rcv_2(UNSIGNED argc, VOID *argv);
void entry_send(UNSIGNED argc, VOID *argv);
```

Application_Initialize will be used to create the dynamic memory pool, out of which memory will be allocated for the three tasks in the system. Application_Initialize will also be used to create the mailbox which will be used to communicate between the three tasks in the system.

```
void Application_Initialize(VOID *first_available_memory)
{
```

Create the dynamic memory pool, and associate it with the dm_memory control block. The memory pool will be 43008 bytes large, will start at first_available_memory, and, if memory is unavailable, tasks that choose to suspend will be resumed in First-In-First-Out order. The minimum allocation from this pool will be 128 bytes. For more information on the NU_Create_Memory_Pool call, or dynamic memory pools in general, see Chapter 4.



```
NU_Create_Memory_Pool(&dm_memory, "sysmem", first_available_memory,
                     43008, 128, NU_FIFO);
```

For each task in the system, allocate 1024 bytes of memory for their respective stacks. With the `NU_Allocate_Memory` call, we are allocating a 1024 byte block of memory out of the `dm_memory` dynamic memory pool. A pointer to the newly allocated memory is assigned to `stack_rcv_1`, `stack_rcv_2`, and `stack_send` respectively. The pointer to this memory allocation is passed to the `NU_Create_Task` call, which will use this memory as the task stack.

For this demonstration, note that `task_rcv_1` and `task_rcv_2` are given a higher priority (priority level of 7) than `task_send`. By doing this, we are ensuring that `task_rcv_1` and `task_rcv_2` will always run before `task_send`. The `task_send` will only run when both `task_rcv_1` and `task_rcv_2` are suspended.

```
NU_Allocate_Memory(&dm_memory, &stack_rcv_1, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_rcv_1, "rcv_1", entry_rcv_1, 0, NU_NULL,
              stack_rcv_1, 1024, 7, 0, NU_PREEMPT, NU_START);

NU_Allocate_Memory(&dm_memory, &stack_rcv_2, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_rcv_2, "rcv_2", entry_rcv_2, 0, NU_NULL,
              stack_rcv_2, 1024, 7, 0, NU_PREEMPT, NU_START);

NU_Allocate_Memory(&dm_memory, &stack_send, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_send, "send", entry_send, 0, NU_NULL,
              stack_send, 1024, 8, 0, NU_PREEMPT, NU_START);
```

Use `NU_Create_Mailbox` to create the `mailbox_comm` mailbox. This mailbox will be named “comm”, and tasks that choose to suspend on this mailbox will be resumed in First-In-First-Out order. Instead of specifying `NU_FIFO`, `NU_PRIORITY` could be specified instead, which would cause tasks to be resumed based upon their priority. For this example, the only tasks that will be suspending on this mailbox are of the same priority, so the results will be the same regardless of the suspension type specified.

```
NU_Create_Mailbox(&mailbox_comm, "comm", NU_FIFO);
}
```

The `entry_rcv_1` and `entry_rcv_2` functions serve as the entry point for the `task_rcv_1` and `task_rcv_2` tasks respectively. The tasks will continuously loop, issuing `NU_Receive_From_Mailbox` for each iteration of the loop. `NU_Receive_From_Mailbox` will suspend until there is a message placed into the mailbox (as indicated by `NU_SUSPEND`). Whenever a message is received, `NU_Receive_From_Mailbox` will exit with a return value of `NU_SUCCESS`. After the call has returned, `rcvmsg` will contain the message received. Therefore, there are two tasks that are continuously suspending on the same mailbox, both waiting for a message to be placed into the mailbox.

The PLUS scheduler will resume these tasks based on the `suspend_type` flag that was specified when the `mailbox_comm` message box was created.

```
void entry_rcv_1(UNSIGNED argc, VOID *argv)
{
    UNSIGNED rcvmsg[4];
```




```

while(1)
{
    if (NU_Receive_From_Mailbox(&mailbox_comm, recvmmsg, NU_SUSPEND)
        == NU_SUCCESS)
    {
        /* recvmmsg contains the received message. */
    }
    else
    {
        /* an error has occurred. */
    }
}
}

```

```

void entry_recv_2(UNSIGNED argc, VOID *argv)
{
    UNSIGNED recvmmsg[4];

    while(1)
    {
        if (NU_Receive_From_Mailbox(&mailbox_comm, recvmmsg, NU_SUSPEND)
            == NU_SUCCESS)
        {
            /* recvmmsg contains the received message. */
        }
        else
        {
            /* an error has occurred. */
        }
    }
}

```

The function `entry_send` serves as the task entry point for the `task_send` task. Note that the `task_recv_1` and `task_recv_2` tasks are of a higher priority, and will always be given first chance to run. Because of this, whenever `task_send` sends a message with the `mailbox_comm` message box, either `task_recv_1` or `task_recv_2` will be immediately resumed.

The `task_send` task continuously loops, and for each iteration of the loop it makes calls to two different PLUS services. The first service call is to `NU_Send_To_Mailbox` which will send a single message with the `mailbox_comm` mailbox. The second service call that is issued is `NU_Broadcast_To_Mailbox`, which will send the message to every task that is currently suspended on this mailbox. Note that in this example, whenever this task is running, there will always be two tasks (`task_recv_1` and `task_recv_2`) suspended on the `mailbox_comm` mailbox. The result is that the message that is sent with `NU_Send_To_Mailbox` will only be received by one of the suspended tasks, while the message sent with `NU_Broadcast_To_Mailbox` will be received by both suspended tasks.

```

void entry_send(UNSIGNED argc, VOID *argv)
{
    UNSIGNED sendmmsg[4];

    while(1)
    {

```

Place decimal 1 in the first element of the four-element array, then issue `NU_Send_To_Mailbox` on the `mailbox_comm` message box. Since two tasks will always be suspended on this mailbox, and the mailbox was created with the `NU_FIFO` suspension flag, the first task that suspended on the mailbox will always receive this message.

```
sendmsg[0]=1;
if (NU_Send_To_Mailbox(&mailbox_comm, sendmsg, NU_SUSPEND)
== NU_SUCCESS)
{
/* The message was successfully sent. */
}
else
{
/* An error occurred, or the message could not be sent. */
}
}
```

Place a decimal 2 in the first element of the four-element array, then issue `NU_Broadcast_To_Mailbox` on the `mailbox_comm` message box. Because the priority of `task_rcv_1` and `task_rcv_2` is higher than this task, we are guaranteed that two tasks will always be suspended on this mailbox. Therefore, the result of the `NU_Broadcast_To_Mailbox` service is that both tasks will be sent the message.

```
sendmsg[0]=2;
if (NU_Broadcast_To_Mailbox(&mailbox_comm, sendmsg, NU_SUSPEND)
== NU_SUCCESS)
{
/* The message was successfully sent. */
}
else
{
/* An error occurred, or the message could not be sent. */
}
}
}
```



7

Queues

Introduction

Function Reference

Example Source Code

Introduction

Queues provide a mechanism to transmit multiple messages. Messages are sent and received by value. A send-message request copies the message into the queue, while a receive-message request copies the message out of the queue. Messages may be placed at the front of the queue or at the back of the queue.

Message Size

A queue message consists of one or more 32-bit words. Both fixed and variable-length messages are supported. The type of message format is defined when the queue is created. Variable-length message queues require an additional 32-bit word of overhead for each message in the queue. Additionally, receive message requests on variable-length message queues specify the *maximum* message size, while the same requests on fixed-length message queues specify the *exact* message size.

Suspension

Send and receive queue services provide options for unconditional suspension, suspension with a timeout, and no suspension.

Tasks may suspend on a queue for several reasons. A task attempting to receive a message from an empty queue can suspend. Additionally, a task attempting to send a message to a full queue can suspend. A suspended task is resumed when the queue is able to satisfy that task's request. For example, suppose a task is suspended on a queue waiting to receive a message. When a message is sent to the queue, the suspended task is resumed.

Multiple tasks may suspend on a single queue. Tasks are suspended in either FIFO or priority order, depending on how the queue was created. If the queue supports FIFO suspension, tasks are resumed in the order in which they were suspended. Otherwise, if the queue supports priority suspension, tasks are resumed from high priority to low priority.

Broadcast

A queue message may be broadcast. This service is similar to a send request, except that all tasks waiting for a message from the queue are given the broadcast message.

Dynamic Creation

Nucleus PLUS queues are created and deleted dynamically. There is no preset limit on the number of queues an application may have. Each queue requires a control block and a queue data area. The memory for each is supplied by the application.

Determinism

Basic processing time required for sending and receiving queue messages is constant. However, the time required to copy a message is relative to the size of the message.



Additionally, processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the queue.

Queue Information

Application tasks may obtain a list of active queues. Detailed information about each queue can also be obtained. This information includes the queue name, message format, suspension type, number of messages present, and the first task waiting.

Function Reference

The following function reference contains all functions related to Nucleus PLUS queues. The following functions are contained in this reference:

- NU_Broadcast_To_Queue
- NU_Create_Queue
- NU_Delete_Queue
- NU_Established_Queues
- NU_Queue_Information
- NU_Queue_Pointers
- NU_Receive_From_Queue
- NU_Reset_Queue
- NU_Send_To_Front_Of_Queue
- NU_Send_To_Queue



NU_Broadcast_To_Queue

```
STATUS NU_Broadcast_To_Queue(NU_QUEUE *queue, VOID *message,
                             UNSIGNED size, UNSIGNED suspend)
```

This service broadcasts a message to all tasks waiting for a message from the specified queue. If no tasks are waiting, the message is simply placed at the end of the queue. Queues are capable of holding multiple messages. Queue messages are comprised of a fixed or variable number of `UNSIGNED` data elements, depending on how this queue was created. The parameters of this service are further defined as follows:

Overview

Option	
Tasking Changes	Yes
Allowed From	<code>Application_Initialize</code> , HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
<code>queue</code>	Pointer to the user-supplied queue control block.
<code>message</code>	Pointer to the broadcast message.
<code>size</code>	Specifies the number of <code>UNSIGNED</code> data elements in the message. If the queue supports variable-length messages, this parameter must be equal to or less than the message size supported by the queue. If the queue supports fixed-size messages, this parameter must be exactly the same as the message size supported by the queue.
<code>suspend</code>	Specifies whether or not to suspend the calling task if there is insufficient room in the queue to hold the message.

Suspension

The following table summarizes the possible values for the `suspend` parameter.

Suspension Option	Meaning
<code>NU_NO_SUSPEND</code>	The service returns immediately regardless of whether or not the request can be satisfied. Note: this is the only valid option if the service is called from a non-task thread.
<code>NU_SUSPEND</code>	The calling task is suspended until the message can be copied into the queue.
<code>timeout value</code>	(1 – 4,294,967,293). The calling task is suspended until the message can be copied into the queue or until the specified number of ticks has expired.



Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_QUEUE	Indicates the queue pointer is invalid.
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_SIZE	Indicates that the message size specified is not compatible with the size specified when the queue was created.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_QUEUE_FULL	Indicates the message could not be immediately placed in the queue because there was not enough space available.
NU_TIMEOUT	Indicates that the queue is still unable to accept the message even after suspending for the specified timeout value.
NU_QUEUE_DELETED	Queue was deleted while the task was suspended.
NU_QUEUE_RESET	Queue was reset while the task was suspended.

Example

```

NU_QUEUE    Queue;
UNSIGNED    message[4];
STATUS      status
.
.
.
/* Build a message to send to a queue.  The contents of
   "message" are not significant. */

message[0]   = 0x00001111;
message[1]   = 0x22223333;
message[3]   = 0x44445555;
message[4]   = 0x66667777;

/* Send the message to the queue control block "Queue".
   If the queue is full, suspend until the request can
   be satisfied.  Assume "Queue" has previously been
   created with the Nucleus PLUS NU_Create_Queue
   service call.*/

status = NU_Broadcast_To_Queue(&Queue, &message[0], 4,
                               NU_SUSPEND);

/* At this point, status indicates whether the service
   request was successful.  */

```

See Also

```

NU_Send_To_Queue, NU_Send_To_Front_Of_Queue, NU_Receive_From_Queue,
NU_Queue_Information

```



NU_Create_Queue

```
STATUS NU_Create_Queue(NU_QUEUE *queue, char *name,
                      VOID *start_address,
                      UNSIGNED queue_size,
                      OPTION message_type,
                      UNSIGNED message_size,
                      OPTION suspend_type)
```

This service creates a message queue. Queues are created to support management of either fixed or variable sized messages. Queue messages are comprised of one or more `UNSIGNED` data elements. The parameters of this service are further defined as follows:

Overview

Option	
Tasking Changes	No
Allowed From	<code>Application_Initialize</code> , <code>HISR</code> , <code>Signal Handler</code> , <code>Task</code>
Category	Task Communication Services

Parameters

Parameter	Meaning
<code>queue</code>	Pointer to the user-supplied queue control block. Note: all subsequent requests made to the queue require this pointer.
<code>name</code>	Pointer to an 8 character name for the queue. The name does not have to be null-terminated.
<code>start_address</code>	Specifies the starting address for the queue. Note: this address must be properly aligned for <code>UNSIGNED</code> data access.
<code>queue_size</code>	Specifies the number of <code>UNSIGNED</code> elements in the queue.
<code>message_type</code>	Specifies the type of messages managed by the queue. <code>NU_FIXED_SIZE</code> specifies that the queue manages fixed-size messages. Note: a fixed-size message queue only uses the area of the queue that is evenly divisible by the message size. <code>NU_VARIABLE_SIZE</code> indicates that the queue manages variable-size messages. Note: each variable-size message requires an additional <code>UNSIGNED</code> data element of overhead inside the queue.
<code>message_size</code>	If the queue supports fixed-size messages, this parameter specifies the exact size of each message. Otherwise, if the queue supports variable-size messages, this parameter indicates the maximum message size. All sizes are in terms of <code>UNSIGNED</code> data elements.
<code>suspend_type</code>	Specifies how tasks suspend on the queue. Valid options for this parameter are <code>NU_FIFO</code> and <code>NU_PRIORITY</code> , which represent First-In-First-Out (FIFO) and priority-order task suspension, respectively.

Return Value



Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_QUEUE	Indicates the queue control block pointer is NULL or is already in use.
NU_INVALID_MEMORY	Indicates the memory area specified by the start_address or size parameters is invalid.
NU_INVALID_MESSAGE	Indicates that the message_type parameter is invalid.
NU_INVALID_SIZE	Indicates that the message size is greater than the queue size or that the message size is zero.
NU_INVALID_SUSPEND	Indicates the suspend_type parameter is invalid.

Example

```

/* Assume queue control block "Queue" is defined as
   a global data structure. This is one of several
   ways to allocate a control block. */
NU_QUEUE Queue;
.
.
/* Assume status is defined locally. */

STATUS      status;    /* Queue creation status */

/* Create a queue with a capacity of 1000 UNSIGNED
   elements starting at the address pointed to by the
   variable "start." Variable-length messages are
   supported, with a maximum message size of 20. Tasks
   suspend on this queue in FIFO order. */

status =  NU_Create_Queue(&Queue, "any name", start,      1000,
                        NU_VARIABLE_SIZE, 20,  NU_FIFO);

/* At this point status indicates if the service was
   successful. */

```

See Also

NU_Delete_Queue, NU_Established_Queues, NU_Queue_Pointers,
 NU_Queue_Information, NU_Reset_Queue



NU_Delete_Queue

```
STATUS NU_Delete_Queue(NU_QUEUE *queue)
```

This service deletes a previously created message queue. The parameter `queue` identifies the message queue to delete. Tasks suspended on this queue are resumed with the appropriate error status. The application must prevent the use of this queue during and after deletion.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
<code>queue</code>	Pointer to the user-supplied queue control block.

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_QUEUE</code>	Indicates the queue pointer is invalid.

Example

```
NU_QUEUE    Queue;
STATUS      status
.
.
.
/* Delete the queue control block "Queue". Assume "Queue"
   has previously been created with the Nucleus PLUS
   NU_Create_Queue service call. */
status = NU_Delete_Queue(&Queue);

/* At this point, status indicates whether the service
   request was successful. */
```

See Also

`NU_Create_Queue`, `NU_Established_Queues`, `NU_Queue_Pointers`,
`NU_Queue_Information`, `NU_Reset_Queue`



NU_Established_Queues

UNSIGNED NU_Established_Queues(VOID)

This service returns the number of established queues. All created queues are considered established. Deleted queues are no longer considered established.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

None

Return Value

This service call returns the number of created queues in the system

Example

```

UNSIGNED total_queues;

/* Obtain the number of queues. */
total_queues = NU_Established_Queues();

```

See Also

NU_Create_Queue, NU_Delete_Queue, NU_Queue_Pointers,
 NU_Queue_Information, NU_Reset_Queue

NU_Queue_Information

```
STATUS NU_Queue_Information(NU_QUEUE *queue, CHAR *name,
                           VOID **start_address,
                           UNSIGNED *queue_size,
                           UNSIGNED *available,
                           UNSIGNED *messages,
                           OPTION *message_type,
                           UNSIGNED *message_size,
                           OPTION *suspend_type,
                           UNSIGNED *tasks_waiting,
                           NU_TASK **first_task)
```

This service returns various information about the specified message-communication queue.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services



Parameters

Parameter	Meaning
queue	Pointer to the user-supplied queue control block.
name	Pointer to an 8 character destination area for the message-queue's name.
start_address	Pointer to a memory pointer for holding the starting address of the queue.
queue_size	Pointer to a variable for holding the total number of <code>UNSIGNED</code> data elements in the queue.
available	Pointer to a variable for holding the number of available <code>UNSIGNED</code> data elements in the queue.
messages	Pointer to a variable for holding the number of messages currently in the queue.
message_type	Pointer to a variable for holding the type of messages supported by the queue. Valid message types are <code>NU_FIXED_SIZE</code> and <code>NU_VARIABLE_SIZE</code> .
message_size	Pointer to a variable for holding the number of <code>UNSIGNED</code> data elements in each queue message. If the queue supports variable-length messages, this number is the maximum message size.
suspend_type	Pointer to a variable for holding the task suspend type. Valid task suspend types are <code>NU_FIFO</code> and <code>NU_PRIORITY</code> .
tasks_waiting	Pointer to a variable for holding the number of tasks waiting on the queue.
first_task	Pointer to a task pointer. The pointer of the first suspended task is placed in this task pointer.

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_QUEUE</code>	Indicates the queue pointer is invalid.

Example

```

NU_QUEUE   Queue;
CHAR        queue_name[8];
VOID        *start_address;
UNSIGNED    size;
UNSIGNED    available;
UNSIGNED    messages;
OPTION      message_type;
UNSIGNED    message_size;
OPTION      suspend_type;
UNSIGNED    tasks_suspended;
NU_TASK     *first_task;
STATUS      status;

/* Obtain information about the message queue control
   block "Queue". Assume "Queue" has previously been
   created with the Nucleus PLUS NU_Create_Queue service
   call. */
status = NU_Queue_Information(&Queue, queue_name, &start_address,
                             &size, &available, &messages,
                             &message_type, &message_size,
                             &suspend_type, &tasks_suspended,
                             &first_task);

/* If status is NU_SUCCESS, the other information is accurate. */

```

See Also

NU_Create_Queue, NU_Delete_Queue, NU_Established_Queues,
 NU_Queue_Pointers, NU_Reset_Queue



NU_Queue_Pointers

```
UNSIGNED NU_Queue_Pointers(NU_QUEUE **pointer_list,
                           UNSIGNED maximum_pointers)
```

This service builds a sequential list of pointers to all established message queues in the system. **Note:** queues that have been deleted are no longer considered established. The parameter `pointer_list` points to the location for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
<code>pointer_list</code>	Pointer to an array of <code>NU_QUEUE</code> pointers. This array will be filled with pointers of established queues in the system.
<code>maximum_pointers</code>	The maximum number of <code>NU_QUEUE</code> pointers to place into the array. Typically this will be the size of the <code>pointer_list</code> array.

Return Value

This service call returns the number of created queues in the system.

Example

```
/* Define an array capable of holding 20 queue pointers. */
NU_QUEUE *Pointer_Array[20];
UNSIGNED number;

/* Obtain a list of currently active queue pointers
   (Maximum of 20). */
number = NU_Queue_Pointers(&Pointer_Array[0],20);

/* At this point, number contains the actual number
   of pointers in the list. */
```

See Also

`NU_Create_Queue`, `NU_Delete_Queue`, `NU_Established_Queues`,
`NU_Queue_Information`, `NU_Reset_Queue`



NU_Receive_From_Queue

```
STATUS NU_Receive_From_Queue(NU_QUEUE *queue, VOID *message,
                             UNSIGNED size, UNSIGNED *actual_size,
                             UNSIGNED suspend)
```

This service retrieves a message from the specified queue. If the queue contains one or more messages, the message in front is immediately removed from the queue and copied into the designated location. Queue messages are comprised of a fixed or variable number of `UNSIGNED` data elements, depending on the type of messages supported by the queue.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
queue	Pointer to the user-supplied queue control block.
message	Pointer to the message destination. Note: the message destination must be capable of holding "size" <code>UNSIGNED</code> data elements.
size	Specifies the number of <code>UNSIGNED</code> data elements in the message. This number must correspond to the message size defined when the queue was created. Only applies to queues defined with fixed message size; otherwise ignored.
actual_size	Pointer to a variable to hold the actual number of <code>UNSIGNED</code> data elements in the received message.
suspend	Specifies whether or not to suspend the calling task if the queue is empty.



Suspension

The following table summarizes the possible values for the `suspend` parameter.

Suspension Option	Meaning
NU_NO_SUSPEND	The service returns immediately regardless of whether or not the request can be satisfied. Note: this is the only valid option if the service is called from a non-task thread.
NU_SUSPEND	The calling task is suspended until a message is available.
timeout value	(1 – 4,294,967,293). The calling task is suspended until a message is available or until the specified number of ticks has expired.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_QUEUE	Indicates the queue pointer is invalid.
NU_INVALID_POINTER	Indicates the message destination pointer is <code>NULL</code> or the “actual_size” pointer is <code>NULL</code> .
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_QUEUE_EMPTY	Indicates the queue is empty.
NU_INVALID_SIZE	Indicates the <code>size</code> parameter is different from the message size supported by the queue. Applies only to queues defined with fixed message size.
NU_TIMEOUT	Indicates that the queue is still empty even after suspending for the specified timeout value.
NU_QUEUE_DELETED	Queue was deleted while the task was suspended.
NU_QUEUE_RESET	Queue was reset while the task was suspended.

Example

```
NU_QUEUE      Queue;
UNSIGNED      message[4];
UNSIGNED      actual_size;
STATUS        status;
.
.
.
/* Receive a 4-UNSIGNED data element message from the
   queue control block "Queue".  If the queue is empty,
   suspend until the request can be satisfied. Assume
   "Queue" has previously been created with the Nucleus
   PLUS NU_Create_Queue service call. */
status = NU_Receive_From_Queue(&Queue, &message[0], 4,
                               &actual_size, NU_SUSPEND);

/* At this point, status indicates whether the service
   request was successful. If successful, "message"
   contains the received message. */
```

See Also

NU_Broadcast_To_Queue, NU_Send_To_Queue, NU_Send_To_Front_Of_Queue,
NU_Queue_Information



NU_Reset_Queue

```
STATUS NU_Reset_Queue (NU_QUEUE *queue)
```

This service discards all messages currently in the queue specified by `queue`.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
<code>queue</code>	Pointer to the user-supplied queue control block.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_QUEUE	Indicates the queue pointer is invalid.

Example

```
NU_QUEUE Queue;
STATUS status
.
.
.
/* Reset the queue control block "Queue". Assume "Queue"
   has previously been created with the Nucleus PLUS
   NU_Create_Queue service call. */
status = NU_Reset_Queue(&Queue);
```

See Also

NU_Broadcast_To_Queue, NU_Send_To_Queue, NU_Send_To_Front_Of_Queue,
NU_Receive_From_Queue, NU_Queue_Information



NU_Send_To_Front_Of_Queue

```
STATUS NU_Send_To_Front_Of_Queue(NU_QUEUE *queue,
                                VOID *message,
                                UNSIGNED size,
                                UNSIGNED suspend)
```

This service places a message at the front of the specified queue. If there is enough space in the queue to hold the message, this service is processed immediately. Queue messages are comprised of a fixed or variable number of `UNSIGNED` data elements, depending on the types of messages supported by the queue.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
queue	Pointer to the user-supplied queue control block.
message	Pointer to the message to send.
size	Specifies the number of <code>UNSIGNED</code> data elements in the message. If the queue supports variable-length messages, this parameter must be equal to or less than the same as the message size supported by the queue.
suspend	Specifies whether or not to suspend the calling task if the queue is full.

Suspension

The following table summarizes the possible values for the `suspend` parameter.

Suspension Option	Meaning
NU_NO_SUSPEND	The service returns immediately regardless of whether or not the request can be satisfied. Note: this is the only valid option if the service is called from a non-task thread.
NU_SUSPEND	The calling task is suspended until the message can be sent.
timeout value	(1 – 4,294,967,293). The calling task is suspended until the message can be sent or until the specified number of ticks has expired.



Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_QUEUE	Indicates the queue pointer is invalid.
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_SIZE	Indicates the specified message size is incompatible with the message size supported by the queue.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_QUEUE_FULL	Indicates the queue is full.
NU_TIMEOUT	Indicates that the queue is still full even after suspending for the specified timeout value.
NU_QUEUE_DELETED	Queue was deleted while the task was suspended.
NU_QUEUE_RESET	Queue was reset while the task was suspended.

Example

```

NU_QUEUE    Queue;
UNSIGNED    message[4];
STATUS      status
.
.
.
/* Build a 4 UNSIGNED variable message to send. The contents
   of "message" have no significance. */
message[0]   = 0x00001111;
message[1]   = 0x00002222;
message[2]   = 0x00003333;
message[3]   = 0x00004444;

/* Send message to the queue control block "Queue". Suspend
the calling task until the message can be sent. Assume
"Queue" has previously been created with the Nucleus PLUS
NU_Create_Queue service call. */
Status = NU_Send_To_Front_Of_Queue(&Queue, &message[0],
                                   4, NU_SUSPEND);

/* At this point, status indicates whether the service
   request was successful. If successful, "message" was
   sent to "Queue". */

```

See Also

```

NU_Broadcast_To_Queue, NU_Receive_From_Queue, NU_Send_To_Queue,
NU_Queue_Information

```



NU_Send_To_Queue

```
STATUS NU_Send_To_Queue(NU_QUEUE *queue, VOID *message,
                        UNSIGNED size, UNSIGNED suspend)
```

This service places a message at the back of the specified queue. If there is enough space in the queue to hold the message, this service is processed immediately. Queue messages are comprised of a fixed or variable-number of `UNSIGNED` data elements, depending on the type of messages supported by the queue.

Overview

Option	
Tasking Changes	Yes
Allowed From	<code>Application_Initialize</code> , HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
<code>queue</code>	Pointer to the user-supplied queue control block.
<code>message</code>	Pointer to the message to send.
<code>size</code>	Specifies the number of <code>UNSIGNED</code> data elements in the message. If the queue supports variable-length messages, this parameter must be equal to or less than the message size supported by the queue. If the queue supports fixed-size messages, this parameter must be exactly the same as the message size supported by the queue.
<code>suspend</code>	Specifies whether or not to suspend the calling task if the queue is full.

Suspension

The following table summarizes the possible values for the `suspend` parameter.

Suspension Option	Meaning
<code>NU_NO_SUSPEND</code>	The service returns immediately regardless of whether or not the request can be satisfied. Note: this is the only valid option if the service is called from a non-task thread.
<code>NU_SUSPEND</code>	The calling task is suspended until the message can be sent.
<code>timeout value</code>	(1 – 4,294,967,293). The calling task is suspended until the message can be sent or until the specified number of ticks has expired.



Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_QUEUE	Indicates the queue pointer is invalid.
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_SIZE	Indicates the message size is incompatible with the message size supported by the queue.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_QUEUE_FULL	Indicates the queue is full.
NU_TIMEOUT	Indicates that the queue is still full even after suspending for the specified timeout value.
NU_QUEUE_DELETED	Queue was deleted while the task was suspended.
NU_QUEUE_RESET	Queue was reset while the task was suspended.

Example

```

NU_QUEUE    Queue;
UNSIGNED    message[4];
STATUS      status;
.
.
.
/* Build a 4 UNSIGNED variable message to send.
   The contents of "message" have no significance. */
message[0]   = 0x00001111;
message[1]   = 0x00002222;
message[2]   = 0x00003333;
message[3]   = 0x00004444;

/* Send the message to the queue control block "Queue".
   Suspend the calling task until the message can be
   sent. Assume "Queue" has previously been created
   with the Nucleus PLUS NU_Create_Queue service call. */
status = NU_Send_To_Queue(&Queue, &message[0], 4, NU_SUSPEND);

/* At this point, status indicates whether the service
   request was successful. If successful, "message"
   was sent to "Queue". */

```

See Also

```

NU_Broadcast_To_Queue, NU_Receive_From_Queue,
NU_Send_To_Front_Of_Queue, NU_Queue_Information

```



Example Source Code

In the previous chapter we looked at an example that demonstrated how to communicate between tasks with mailboxes. In this section we will look at a very similar example, but using queues to communicate between several tasks.

Include all necessary Nucleus PLUS include files.

```
#include "nucleus.h"
```

Five Nucleus PLUS structures are used in this example. Three `NU_TASK` structures are used, one for each task in the system. The `NU_QUEUE` structure is for the queue that will be used to communicate messages between the three tasks in the system. An `NU_MEMORY_POOL` structure is also used to allocate any memory, which in this example is for the queue data area and a stack for each of the three tasks.

```
NU_TASK task_recv_1;
NU_TASK task_recv_2;
NU_TASK task_send;
NU_QUEUE queue_comm;
NU_MEMORY_POOL dm_memory;
```

The three void pointers `stack_recv_1`, `stack_recv_2`, and `stack_send` will each hold a pointer to a separate task stack. Although not demonstrated in this program, these pointers could be used at a later time in the program to deallocate the task stacks, or they could be discarded if the task stacks will never be deallocated.

```
VOID *stack_recv_1;
VOID *stack_recv_2;
VOID *stack_send;
```

Similar to the above three void pointers, the `data_queue` pointer will be used to hold a pointer to the data area for the queue. It can either be used to deallocate the associated memory, or discarded if memory deallocation is not necessary.

```
VOID *data_queue;
```

Declare the task entry point function for each of the three tasks. These will later be passed as a parameter to the `NU_Create_Task` call which will associate these functions with each of their respective tasks.

```
void entry_recv_1(UNSIGNED argc, VOID *argv);
void entry_recv_2(UNSIGNED argc, VOID *argv);
void entry_send(UNSIGNED argc, VOID *argv);
```



`Application_Initialize` will be used to create the dynamic memory pool, out of which memory will be allocated for three task stacks, and the queue data area. Therefore, in `Application_Initialize` there are four separate calls to `NU_Allocate_Memory`. `Application_Initialize` is also used to create the queue and associate the allocated memory for its queue data area.

```
void Application_Initialize(VOID *first_available_memory)
{
```

Create the dynamic memory pool and associate it with the `dm_memory` control block. The memory pool will 43008 bytes large, will start at `first_available_memory`, and, if memory is unavailable, tasks that choose to suspend will resume in First-In-First-Out order. The minimum allocation from this pool will be 128 bytes. For more information on the `NU_Create_Memory_Pool` call, or dynamic memory pools in general, see Chapter 4.

```
NU_Create_Memory_Pool(&dm_memory, "system", first_available_memory,
                     43008, 128, NU_FIFO);
```

For each task in the system, allocate 1024 bytes of memory for their respective stacks. With the `NU_Allocate_Memory` call, we are allocating a 1024 byte block of memory out of the `dm_memory` dynamic memory pool. A pointer to the newly allocated memory is assigned to the `stack_rcv_1`, `stack_rcv_2`, and `stack_send` respectively. The pointer to this memory allocation is passed to the `NU_Create_Task` call, which will use this memory as the task stack.

For this demonstration, note that `task_rcv_1` and `task_rcv_2` are given a higher priority (priority level of 7) than `task_send`. By doing this, we are ensuring that `task_rcv_1` and `task_rcv_2` will always run before `task_send`. The `task_send` will only run when both `task_rcv_1` and `task_rcv_2` are suspended.

```
NU_Allocate_Memory(&dm_memory, &stack_rcv_1, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_rcv_1, "rcv_1", entry_rcv_1, 0, NU_NULL,
              stack_rcv_1, 1024, 7, 0, NU_PREEMPT, NU_START);

NU_Allocate_Memory(&dm_memory, &stack_rcv_2, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_rcv_2, "rcv_2", entry_rcv_2, 0, NU_NULL,
              stack_rcv_2, 1024, 7, 0, NU_PREEMPT, NU_START);

NU_Allocate_Memory(&dm_memory, &stack_send, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_send, "send", entry_send, 0, NU_NULL,
              stack_send, 1024, 8, 0, NU_PREEMPT, NU_START);
```

First, allocate memory for the queue data area with a call to `NU_Allocate_Memory`. This call allocates 32768 bytes out of the `dm_memory` dynamic memory pool, and assigns a pointer to this memory to the `data_queue` void pointer. Then call `NU_Create_Queue` to associate this memory to the `queue_comm` queue. The `queue_comm` queue is a queue with fixed sized messages (`NU_FIXED_SIZE`), and each message will be 32-bits in size. The queue is associated with the name “comm” and tasks that choose to suspend on this queue will be resumed in First-In-First-Out order.

```
NU_Allocate_Memory(&dm_memory, &data_queue, 32768, NU_NO_SUSPEND);
NU_Create_Queue(&queue_comm, "comm", data_queue, 32768,
               NU_FIXED_SIZE, 1, NU_FIFO);
}
```

The `entry_rcv_1` and `entry_rcv_2` functions serve as the entry point for the `task_rcv_1` and `task_rcv_2` tasks respectively. The tasks will continuously loop, issuing an `NU_Receive_From_Queue` call for each iteration of the loop. The `NU_Receive_From_Queue` will suspend until there is a message placed into the queue (as indicated by `NU_SUSPEND`). Whenever a message is received, `NU_Receive_From_Queue` will exit with a return value of `NU_SUCCESS`. After the call has returned, `rcvmsg` will contain the message received. Therefore, there are two tasks that are continuously suspending on the same queue, both waiting for a message to be placed into the queue. The PLUS scheduler will resume these tasks based on the `suspend_type` flag that was specified when the `queue_comm` queue was created.

```
void entry_rcv_1(UNSIGNED argc, VOID *argv)
{
    UNSIGNED rcvmsg;
    UNSIGNED actual_size;

    while(1)
    {
        if (NU_Receive_From_Queue(&queue_comm, &rcvmsg, 1, &actual_size,
                                NU_SUSPEND) == NU_SUCCESS)
        {
            /* rcvmsg contains the received message. */
        }
        else
        {
            /* an error has occurred. */
        }
    }
}
```



Queues

```

void entry_rcv_2(UNSIGNED argc, VOID *argv)
{
    UNSIGNED rcvmsg;
    UNSIGNED actual_size;

    while(1)
    {
        if (NU_Receive_From_Queue(&queue_comm, &rcvmsg, 1, &actual_size,
                                NU_SUSPEND) == NU_SUCCESS)
        {
            /* rcvmsg contains the received message. */
        }
        else
        {
            /* an error has occurred. */
        }
    }
}

```

The function `entry_send` serves as the task entry point for the `task_send` task. Note that the `task_rcv_1` and `task_rcv_2` tasks are of a higher priority, and will always be given first chance to run. Because of this, whenever `task_send` sends a message with `queue_comm`, either `task_rcv_1` or `task_rcv_2` will be immediately resumed.

The `task_send` task continuously loops, and for each iteration of the loop it makes calls to two different PLUS services. The first service call is to `NU_Send_To_Queue` which will send a single message with the `queue_comm` queue. The second service call that is issued is `NU_Broadcast_To_Queue`, which will send the message to every task that is currently suspended on this queue. Note that in this example, whenever this task is running, there will always be two tasks (`task_rcv_1` and `task_rcv_2`) suspended on the `queue_comm` queue. The result is that the message that is sent with `NU_Send_To_Queue` will only be received by one of the suspended tasks, while the message sent with `NU_Broadcast_To_Queue` will be received by both suspended tasks.

```

void entry_send(UNSIGNED argc, VOID *argv)
{
    UNSIGNED sendmsg;

    while(1)
    {

```

Assign decimal 1 to `sendmsg`, then issue `NU_Send_To_Queue` on the `queue_comm` queue. Since two tasks will always be suspended on this queue, and the queue was created with the `NU_FIFO` suspension flag, the first task that suspended on the queue will always receive this message.

```
sendmsg=1;
if (NU_Send_To_Queue(&queue_comm, &sendmsg, 1, NU_SUSPEND)
    == NU_SUCCESS)
{
    /* recvmmsg contains the received message. */
}
else
{
    /* an error has occurred. */
}
```

Assign decimal 2 to `sendmsg`, then issue `NU_Broadcast_To_Queue` on the `queue_comm` queue. Because the priority of `task_rcv_1` and `task_rcv_2` is higher priority than this task, we are guaranteed that two tasks will always be suspended on this queue. Therefore, the result of the `NU_Broadcast_To_Queue` service is that both tasks will be sent the message.

```
sendmsg=2;
if (NU_Broadcast_To_Queue(&queue_comm, &sendmsg, 1,
    NU_SUSPEND) == NU_SUCCESS)
{
}
else
{
}
```



8

Pipes

Introduction

Function Reference

Example Source Code

Introduction

Pipes provide a mechanism for transmitting multiple messages. Messages are sent and received by value. A send-message request copies the message into the pipe, while a receive-message request copies the message out of the pipe. Messages may be placed at the front of the pipe or at the back of the pipe.

Message Size

A pipe message consists of one or more bytes. Both fixed and variable-length messages are supported. The type of message format is defined when the pipe is created. Variable-length message pipes require an additional 32-bit word of overhead for each message in the pipe. Additionally, receive-message requests on variable-length message pipes specify the *maximum* message size, while the same request on fixed-length message pipes specify the *exact* message size.

Suspension

Send and receive pipe services provide options for unconditional suspension, suspension with a timeout, and no suspension.

Tasks may suspend on a pipe for several reasons. Tasks attempting to receive a message from an empty pipe can suspend. Also, a task attempting to send a message to a full pipe can suspend. A suspended task is resumed when the pipe is able to satisfy that task's request. For example, suppose a task is suspended on a pipe waiting to receive a message. When a message is sent to the pipe, the suspended task is resumed.

Multiple tasks may suspend on a single pipe. Tasks are suspended in either FIFO or priority order, depending on how the pipe was created. If the pipe supports FIFO suspension, tasks are resumed in the order in which they were suspended. Otherwise, if the pipe supports priority suspension, tasks are resumed from high priority to low priority.

Broadcast

A pipe message may be broadcast. This service is similar to a send request, except that all tasks waiting for a message from the pipe are given the broadcast message.

Dynamic Creation

Nucleus PLUS pipes are created and deleted dynamically. There is no preset limit on the number of pipes an application may have. Each pipe requires a control block and a pipe data area. The memory for each is supplied by the application.



Determinism

Basic processing time required for sending and receiving pipe messages is constant. However, the time required to copy a message is relative to the size of the message. Additionally, processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the pipe.

Pipe Information

Application tasks may obtain a list of active pipes. Detailed information about each pipe can also be obtained. This information includes the pipe name, message format, suspension type, number of messages present, and the first task waiting.

Function Reference

The following function reference contains all functions related to Nucleus PLUS pipes. The following functions are contained in this reference:

- NU_Broadcast_To_Pipe
- NU_Create_Pipe
- NU_Delete_Pipe
- NU_Established_Pipes
- NU_Pipe_Information
- NU_Pipe_Pointers
- NU_Receive_From_Pipe
- NU_Reset_Pipe
- NU_Send_To_Front_Of_Pipe
- NU_Send_To_Pipe

NU_Broadcast_To_Pipe

```
STATUS NU_Broadcast_To_Pipe(NU_PIPE *pipe, VOID *message,
                           UNSIGNED size, UNSIGNED suspend)
```

This service broadcasts a message to all tasks waiting for a message from the specified pipe. If no tasks are waiting, the message is simply placed at the end of the pipe. Pipes are capable of holding multiple messages. Pipe messages are comprised of a fixed or variable number of bytes, depending on how the pipe was created.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
pipe	Pointer to the user-supplied pipe control block.
message	Pointer to the broadcast message.
size	Specifies the number of bytes in the message. If the pipe supports variable-length messages, this parameter must be equal to or less than the message size supported by the pipe. If the pipe supports fixed-size messages, this parameter must be exactly the same as the message size supported by the pipe.
suspend	Specifies whether or not to suspend the calling task if there is insufficient room in the pipe to hold the message.

Suspension

The following table summarizes the possible values for the `suspend` parameter.

Suspension Option	Meaning
NU_NO_SUSPEND	The service returns immediately regardless of whether or not the request can be satisfied. Note: this is the only valid option if the service is called from a non-task thread.
NU_SUSPEND	The calling task is suspended until the message can be copied into the pipe.
timeout value	(1 - 4,294,967,293). The calling task is suspended until the message can be copied into the pipe or until the specified number of ticks has expired.



Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_PIPE	Indicates the pipe pointer is invalid.
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_SIZE	Indicates that the message size specified is not compatible with the size specified when the pipe was created.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_PIPE_FULL	Indicates the message could not be immediately placed in the pipe because there was not enough space available.
NU_TIMEOUT	Indicates that the pipe is still unable to accept the message even after suspending for the specified timeout value.
NU_PIPE_DELETED	Pipe was deleted while the task was suspended.
NU_PIPE_RESET	Pipe was reset while the task was suspended.

Example

```

NU_PIPE      Pipe;
UNSIGNED_CHAR message[4];
STATUS      status
.
.
.
/* Build a 4-byte message to send to a pipe. The
   contents of "message" are not significant. */

message[0]   = 0x01;
message[1]   = 0x23;
message[2]   = 0x45;
message[3]   = 0x67;

/* Send a message to the pipe control block "Pipe". Do not
   suspend even if the pipe does not have enough room for
   the message. Assume "Pipe" has previously been created
   with the Nucleus PLUS NU_Create_Pipe service call. */

status = NU_Broadcast_To_Pipe(&Pipe,&message[0], 4,
NU_NO_SUSPEND);

/* At this point, status indicates whether the
   service request was successful. */

```

See Also

```

NU_Send_To_Pipe, NU_Send_To_Front_Of_Pipe, NU_Receive_From_Pipe,
NU_Pipe_Information

```



NU_Create_Pipe

```
STATUS NU_Create_Pipe(NU_PIPE *pipe, CHAR *name,
                     VOID *start_address,
                     UNSIGNED pipe_size,
                     OPTION message_type,
                     UNSIGNED message_size,
                     OPTION suspend_type)
```

This service creates a message pipe. Pipes are created to support management of either fixed or variable sized messages.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
pipe	Pointer to the user-supplied pipe control block. Note: all subsequent requests made to the pipe require this pointer.
name	Pointer to an 8 character name for the pipe. The name does not have to be null-terminated.
start_address	Specifies the starting address for the pipe.
pipe_size	Specifies the total number of bytes in the pipe.
message_type	Specifies the type of messages managed by the pipe.
	NU_FIXED_SIZE Specifies that the pipe manages fixed-size messages. Note: a fixed-size message pipe only uses the area of the pipe that is evenly divisible by the message size.
	NU_VARIABLE_SIZE Indicates that the pipe manages variable-size messages. Note: each variable-size message requires an additional UNSIGNED data type of overhead inside the pipe. Additional padding bytes may be necessary for a message in order to insure UNSIGNED alignment of the next variable-sized message.
message_size	If the pipe supports fixed-size messages, this parameter specifies the exact size of each message. Otherwise, if the pipe supports variable-size messages, this parameter indicates the maximum message size. All sizes are in terms of bytes.
suspend_type	Specifies how tasks suspend on the pipe. Valid options for this parameter are NU_FIFO and NU_PRIORITY, which represent First-In-First-Out (FIFO) and priority-order task suspension, respectively.



Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_PIPE	Indicates the pipe control block pointer is NULL or is already in use.
NU_INVALID_MEMORY	Indicates the memory area specified by the <code>start_address</code> or size parameters is invalid.
NU_INVALID_MESSAGE	Indicates that the <code>message_type</code> parameter is invalid.
NU_INVALID_SIZE	Indicates that the message size specified is larger than the pipe size or is zero.
NU_INVALID_SUSPEND	Indicates the <code>suspend_type</code> parameter is invalid.

Example

```

/* Assume pipe control block "Pipe" is defined as a global
   data structure. This is one of several ways to allocate
   a control block. */

NU_PIPE    Pipe;
.
.
/* Assume status is defined locally. */

STATUS     status;      /* Pipe creation status */

/* Create a pipe in a 1500-byte memory area starting at
   the address pointed to by the variable "start."
   Fixed-size, 20-byte messages are supported by this
   pipe. Tasks suspend on this pipe in order of their
   priority. */

status =  NU_Create_Pipe(&Pipe, "any name", start, 1500,
                        NU_FIXED_SIZE, 20, NU_PRIORITY);

/* At this point status indicates if the service was successful. */

```

See Also

[NU_Delete_Pipe](#), [NU_Established_Pipes](#), [NU_Pipe_Pointers](#),
[NU_Pipe_Information](#), [NU_Reset_Pipe](#)



NU_Delete_Pipe

```
STATUS NU_Delete_Pipe(NU_PIPE *pipe)
```

This service deletes a previously created message pipe. The parameter `pipe` identifies the message pipe to delete. Tasks suspended on this pipe are resumed with the appropriate error status. The application must prevent the use of this pipe during and after deletion.

Overview

Option	
Tasking Changes	Yes
Allowed From	<code>Application_Initialize</code> , <code>HISR</code> , <code>Signal Handler</code> , <code>Task</code>
Category	Task Communication Services

Parameters

Parameter	Meaning
<code>pipe</code>	Pointer to the user-supplied pipe control block.

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_PIPE</code>	Indicates the pipe pointer is invalid.

Example

```
NU_PIPE    Pipe;
STATUS     status
.
.
.
/* Delete the pipe control block "Pipe". Assume
   "Pipe" has previously been created with the Nucleus
   PLUS NU_Create_Pipe service call. */
status =  NU_Delete_Pipe(&Pipe);

/* At this point, status indicates whether the service
   request was successful. */
```

See Also

`NU_Create_Pipe`, `NU_Established_Pipes`, `NU_Pipe_Pointers`,
`NU_Pipe_Information`, `NU_Reset_Pipe`



NU_Established_Pipes

UNSIGNED NU_Established_Pipes(VOID)

This service returns the number of established pipes. All created pipes are considered established. Deleted pipes are no longer considered established.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

None

Return Value

This service call returns the number of created pipes in the system

Example

```
UNSIGNED total_pipes;

/* Obtain the total number of pipes. */
total_pipes = NU_Established_Pipes( );
```

See Also

NU_Create_Pipe, NU_Delete_Pipe, NU_Pipe_Pointers,
NU_Pipe_Information, NU_Reset_Pipe

NU_Pipe_Information

```
STATUS NU_Pipe_Information(NU_PIPE *pipe, CHAR *name,
                          VOID **start_address,
                          UNSIGNED *pipe_size,
                          UNSIGNED *available,
                          UNSIGNED *messages,
                          OPTION *message_type,
                          UNSIGNED *message_size,
                          OPTION *suspend_type,
                          UNSIGNED *tasks_waiting,
                          NU_TASK **first_task)
```

This service returns various information about the specified message-communication pipe.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
pipe	Pointer to the user-supplied pipe control block.
name	Pointer to an 8 character destination area for the pipe's name.
start_address	Pointer for holding the starting address of the pipe.
pipe_size	Pointer for holding the total number of bytes in the pipe.
available	Pointer for holding the number of available bytes in the pipe.
messages	Pointer to a variable for holding the number of messages currently in the pipe.
message_type	Pointer to a variable for holding the type of messages supported by the pipe. Valid message types are <code>NU_FIXED_SIZE</code> and <code>NU_VARIABLE_SIZE</code> .
message_size	Pointer to a variable for holding the number of bytes in each message. If the pipe supports fixed-size messages, this is the exact size of each message. If the pipe supports variable-size messages, this is the maximum size of each message.
suspend_type	Pointer to a variable for holding the task suspend type. Valid task suspend types are <code>NU_FIFO</code> and <code>NU_PRIORITY</code> .
Parameter	Meaning
tasks_waiting	Pointer to a variable for holding the number of tasks waiting on the pipe.
first_task	Pointer to a task pointer. The pointer of the first suspended task is placed in this task pointer.



Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_PIPE	Indicates the pipe pointer is invalid.

Example

```

NU_PIPE    Pipe;
CHAR       pipe_name[8];
VOID       *start_address;
UNSIGNED   pipe_size;
UNSIGNED   available;
UNSIGNED   messages;
OPTION     message_type;
UNSIGNED   message_size;
OPTION     suspend_type;
UNSIGNED   tasks_suspended;
NU_TASK    *first_task;
STATUS     status
.
.
.
/* Obtain information about the message pipe control
   block "Pipe". Assume "Pipe" has previously been
   created with the Nucleus PLUS NU_Create_Pipe service
   call. */
status = NU_Pipe_Information(&Pipe, pipe_name, &start_address,
                           &pipe_size, &available, &messages,
                           &message_type, &message_size,
                           &suspend_type, &tasks_suspended,
                           &first_task);

/* If status is NU_SUCCESS, the other information is accurate. */

```

See Also

NU_Create_Pipe, NU_Delete_Pipe, NU_Established_Pipes,
NU_Pipe_Pointers, NU_Reset_Pipe



NU_Pipe_Pointers

```
UNSIGNED NU_Pipe_Pointers(NU_PIPE **pointer_list,
                          UNSIGNED maximum_pointers)
```

This service builds a sequential list of pointers to all established message pipes in the system. **Note:** pipes that have been deleted are no longer considered established. The parameter `pointer_list` points to the location for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
<code>pointer_list</code>	Pointer to an array of <code>NU_PIPE</code> pointers. This array will be filled with pointers of established pipes in the system.
<code>maximum_pointers</code>	The maximum number of <code>NU_PIPE</code> pointers to place into the array. Typically this will be the size of the <code>pointer_list</code> array.

Return Value

This service call returns the number of created pipes in the system.

Example

```
/* Define an array capable of holding 20 pipe pointers. */
NU_PIPE    *Pointer_Array[20];
UNSIGNED    number;

/* Obtain a list of currently active pipe pointers
   (Maximum of 20). */
number = NU_Pipe_Pointers(&Pointer_Array[0], 20);

/* At this point, number contains the actual number
   of pointers in the list. */
```

See Also

`NU_Create_Pipe`, `NU_Delete_Pipe`, `NU_Established_Pipes`,
`NU_Pipe_Information`, `NU_Reset_Pipe`



NU_Receive_From_Pipe

```
STATUS NU_Receive_From_Pipe(NU_PIPE *pipe,
                           VOID *message,
                           UNSIGNED size,
                           UNSIGNED *actual_size,
                           UNSIGNED suspend)
```

This service retrieves a message from the specified pipe. If the pipe contains one or more messages, the message in front is immediately removed from the pipe and copied into the designated location. Pipe messages are comprised of a fixed or variable number of bytes, depending on the type of the messages supported by the pipe.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
pipe	Pointer to the pipe.
message	Pointer to message destination. Note: the message destination must be large enough to hold <code>size</code> bytes.
size	Specifies the number of bytes in the message. This number must correspond to the message size defined when the pipe was created. Only applies to pipes defined with fixed message size; otherwise ignored.
actual_size	Pointer to a variable to hold the actual number of bytes in the received message.
suspend	Specifies whether or not to suspend the calling task if the pipe is empty.

Suspension

The following table summarizes the possible values for the `suspend` parameter.

Suspension Option	Meaning
NU_NO_SUSPEND	The service returns immediately regardless of whether or not the request can be satisfied. Note: this is the only valid option if the service is called from a non-task thread.
NU_SUSPEND	The calling task is suspended until a message is available.
timeout value	(1 – 4,294,967,293). The calling task is suspended until a message is available or until the specified number of ticks has expired.



Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_PIPE	Indicates the pipe pointer is invalid.
NU_INVALID_POINTER	Indicates the message pointer is <code>NULL</code> or the actual size pointer is <code>NULL</code> .
NU_INVALID_SIZE	Indicates the <code>size</code> parameter is different from the message size supported by the pipe. Applies only to pipes defined with fixed message size.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_PIPE_EMPTY	Indicates the pipe is empty.
NU_TIMEOUT	Indicates that the pipe is still empty even after suspending for the specified timeout value.
NU_PIPE_DELETED	Pipe was deleted while the task was suspended.
NU_PIPE_RESET	Pipe was reset while the task was suspended.

Example

```

NU_PIPE      Pipe;
UNSIGNED_CHAR message[4];
UNSIGNED     actual_size;
STATUS       status;
.
.
.
/* Receive a 4-byte, fixed size message from the pipe
   control block "Pipe". Do not suspend even if the pipe
   is empty. Assume "Pipe" has previously been created
   with the Nucleus PLUS NU_Create_Pipe service call. */
status = NU_Receive_From_Pipe(&Pipe,&message[0], 4, &actual_size,
                             NU_NO_SUSPEND);

/* At this point, status indicates whether the service request
   was successful. If successful, "message" contains the message
   and "actual_size" contains 4.* /

```

See Also

NU_Broadcast_To_Pipe, NU_Send_To_Pipe, NU_Send_To_Front_Of_Pipe,
NU_Pipe_Information



NU_Reset_Pipe

```
STATUS NU_Reset_Pipe(NU_PIPE *pipe)
```

This service discards all messages currently in the pipe specified by `pipe`. All tasks suspended on the pipe are resumed with the appropriate reset status.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
<code>pipe</code>	Pointer to the user-supplied pipe control block.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_PIPE	Indicates the pipe pointer is invalid.

Example

```
NU_PIPE    Pipe;
STATUS     status
.
.
.
/* Reset the pipe control block "Pipe". Assume "Pipe" has
   previously been created with the Nucleus PLUS
   NU_Create_Pipe service call. */
status = NU_Reset_Pipe(&Pipe);
```

See Also

NU_Broadcast_To_Pipe, NU_Send_To_Pipe, NU_Send_To_Front_Of_Pipe,
NU_Receive_From_Pipe, NU_Pipe_Information



NU_Send_To_Front_Of_Pipe

```
STATUS NU_Send_To_Front_Of_Pipe(NU_PIPE *pipe, VOID *message,
                                UNSIGNED size, UNSIGNED suspend)
```

This service places a message at the front of the specified pipe. If there is enough space in the pipe to hold the message, this service is processed immediately. Pipe messages are comprised of a fixed or variable-number of bytes, depending on the type of messages supported by the pipe.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
pipe	Pointer to the pipe.
message	Pointer to the message to send.
size	Specifies the number of bytes in the message. If the pipe supports variable-length messages, this parameter must be equal to or less than the message size supported by the pipe. If the pipe supports fixed-size messages, this parameter must be exactly the same as the message size supported by the pipe.
suspend	Specifies whether or not to suspend the calling task if the pipe is full.

Suspension

The following table summarizes the possible values for the `suspend` parameter.

Suspension Option	Meaning
NU_NO_SUSPEND	The service returns immediately regardless of whether or not the request can be satisfied. Note: this is the only valid option if the service is called from a non-task thread.
NU_SUSPEND	The calling task is suspended until the message can be sent.
timeout value	(1 – 4,294,967,293). The calling task is suspended until the message can be sent or until the specified number of ticks has expired.



Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_PIPE	Indicates the pipe pointer is invalid.
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_SIZE	Indicates the message size is incompatible with the message size supported by the pipe.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_PIPE_FULL	Indicates the pipe is full.
NU_TIMEOUT	Indicates that the pipe is still full even after suspending for the specified timeout value.
NU_PIPE_DELETED	Pipe was deleted while the task was suspended.
NU_PIPE_RESET	Pipe was reset while the task was suspended.

Example

```

NU_PIPE      Pipe;
UNSIGNED_CHAR message[4];
STATUS      status;
.
.
.
/* Build a 4-byte message to send.  The contents of
   "message" have no significance. */
message[0]   = 0x01;
message[1]   = 0x02;
message[2]   = 0x03;
message[3]   = 0x04;

/* Send a 4-byte, fixed size message to the pipe control block
   "Pipe". Do not suspend even if the pipe is full. Assume
   "Pipe" has previously been created with the Nucleus PLUS
   NU_Create_Pipe service call. */
status = NU_Send_To_Front_Of_Pipe(&Pipe, &message[0],
                                   4, NU_NO_SUSPEND);

/* At this point, status indicates whether the service request
   was successful. If successful, "message" was sent to "Pipe". */

```

See Also

```

NU_Broadcast_To_Pipe, NU_Receive_From_Pipe, NU_Send_To_Pipe,
NU_Pipe_Information

```



NU_Send_To_Pipe

```
STATUS NU_Send_To_Pipe(NU_PIPE *pipe, VOID *message,
                      UNSIGNED size, UNSIGNED suspend)
```

This service places a message at the back of the specified pipe. If there is enough space in the pipe to hold the message, this service is processed immediately. Pipe messages are comprised of a fixed or variable-number of bytes, depending on the type of messages supported by the pipe.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Communication Services

Parameters

Parameter	Meaning
pipe	Pointer to the pipe.
message	Pointer to the message to send.
size	Specifies the number of bytes in the message. If the pipe supports variable-length messages, this parameter must be equal to or less than the message size supported by the pipe. If the pipe supports fixed-size messages, this parameter must be exactly the same as the message size supported by the pipe.
suspend	Specifies whether or not to suspend the calling task if the pipe is full.

Suspension

The following table summarizes the possible values for the `suspend` parameter.

Suspension Option	Meaning
NU_NO_SUSPEND	The service returns immediately regardless of whether or not the request can be satisfied. Note: this is the only valid option if the service is called from a non-task thread.
NU_SUSPEND	The calling task is suspended until the message can be sent.
timeout value	(1 – 4,294,967,293). The calling task is suspended until the message can be sent or until the specified number of ticks has expired.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_PIPE	Indicates the pipe pointer is invalid.
NU_INVALID_POINTER	Indicates the message pointer is NULL.
NU_INVALID_SIZE	Indicates the message size is incompatible with the message size supported by the pipe.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_PIPE_FULL	Indicates the pipe is full.
NU_TIMEOUT	Indicates that the pipe is still full even after suspending for the specified timeout value.
NU_PIPE_DELETED	Indicates the pipe was deleted while the task was suspended.
NU_PIPE_RESET	Indicates the pipe was reset while the task was suspended.

Example

```

NU_PIPE          Pipe;
UNSIGNED_CHAR    message[4];
STATUS           status;
.
.
.
/* Build a 4-byte message to send. The contents of
   "message" have no significance. */
message[0] = 0x01;
message[1] = 0x02;
message[2] = 0x03;
message[3] = 0x04;

/* Send a 4-byte message to the pipe control block "Pipe".
   Do not suspend even if the pipe is full. Assume "Pipe"
   has previously been created with the Nucleus
   PLUS NU_Create_Pipe service call. */
status = NU_Send_To_Pipe(&Pipe, &message[0], 4, NU_NO_SUSPEND);

/* At this point, status indicates whether the service
   request was successful. If successful, "message" was
   sent to "Pipe". */

```

See Also

```

NU_Broadcast_To_Pipe, NU_Receive_From_Pipe,
NU_Send_To_Front_Of_Pipe, NU_Pipe_Information

```



Example Source Code

In previous sections we looked at examples that demonstrated how to communicate between tasks with mailboxes and queues. In this section we will look at a very similar example, but using pipes to communicate between several tasks.

Include all necessary Nucleus PLUS include files.

```
#include "nucleus.h"
```

Five Nucleus PLUS structures are used in this example. Three `NU_TASK` structures are used, one for each task in the system. The `NU_PIPE` structure is for the pipe that will be used to communicate messages between the three tasks in the system. An `NU_MEMORY_POOL` structure is also used to allocate any memory, which in this example is for the pipe data area and a stack for each of the three tasks.

```
NU_TASK task_recv_1;
NU_TASK task_recv_2;
NU_TASK task_send;
NU_PIPE pipe_comm;
NU_MEMORY_POOL dm_memory;
```

The three void pointers `stack_recv_1`, `stack_recv_2`, and `stack_send` will each hold a pointer to a separate task stack. Although not demonstrated in this program, these pointers could be used at a later time in the program to deallocate the task stacks, or they could be discarded if the tasks stacks will never be deallocated.

```
VOID *stack_recv_1;
VOID *stack_recv_2;
VOID *stack_send;
```

Similar to the above three void pointers, the `data_pipe` pointer will be used to hold a pointer to the data area for the pipe. It can either be used to deallocate the associated memory, or discarded if memory deallocation is not necessary.

```
VOID *data_pipe;
```

Declare the task entry point function for each of the three tasks. These will later be passed as a parameter to the `NU_Create_Task` call which will associate these functions with each of their respective tasks.

```
void entry_recv_1(UNSIGNED argc, VOID *argv);
void entry_recv_2(UNSIGNED argc, VOID *argv);
void entry_send(UNSIGNED argc, VOID *argv);
```

`Application_Initialize` will be used to create the dynamic memory pool, out of which memory will be allocated for three task stacks, and the pipe data area. Therefore, in `Application_Initialize` there are four separate calls to `NU_Allocate_Memory`. `Application_Initialize` is also used to create the pipe and associate the allocated memory for its pipe data area.

```
void Application_Initialize(VOID *first_available_memory)
{
```



Create the dynamic memory pool and associate it with the `dm_memory` control block. The memory pool will 43008 bytes large, will start at `first_available_memory`, and, if memory is unavailable, tasks that choose to suspend will be resumed in First-In-First-Out order. The minimum allocation from this pool will be 128 bytes. For more information on the `NU_Create_Memory_Pool` call, or dynamic memory pools in general, see Chapter 4.

```
NU_Create_Memory_Pool(&dm_memory, "system", first_available_memory,
                     43008, 128, NU_FIFO);
```

For each task in the system, allocate 1024 bytes of memory for their respective stacks. With the `NU_Allocate_Memory` call, we are allocating a 1024 byte block of memory out of the `dm_memory` dynamic memory pool. A pointer to the newly allocated memory is assigned to the `stack_recv_1`, `stack_recv_2`, and `stack_recv_3` respectively. the pointer to this memory allocation is passed to the `NU_Create_Task` call, which will use this memory as the task stack.

For this demonstration, note that `task_recv_1` and `task_recv_2` are given a higher priority (priority level of 7) than `task_send`. By doing this, we are ensuring that `task_recv_1` and `task_recv_2` will always run before `task_send`. The `task_send` will only run when both `task_recv_1` and `task_recv_2` are suspended.

```
NU_Allocate_Memory(&dm_memory, &stack_recv_1, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_recv_1, "recv_1", entry_recv_1, 0, NU_NULL,
              stack_recv_1, 1024, 7, 0, NU_PREEMPT, NU_START);

NU_Allocate_Memory(&dm_memory, &stack_recv_2, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_recv_2, "recv_2", entry_recv_2, 0, NU_NULL,
              stack_recv_2, 1024, 7, 0, NU_PREEMPT, NU_START);

NU_Allocate_Memory(&dm_memory, &stack_send, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_send, "send", entry_send, 0, NU_NULL,
              stack_send, 1024, 8, 0, NU_PREEMPT, NU_START);
```

Allocate memory for the pipe data area with a call to `NU_Allocate_Memory`. This call allocates 32768 bytes out of the `dm_memory` dynamic memory pool, and assigns a pointer to this memory to the `data_pipe` void pointer. Then call `NU_Create_Pipe` to associate this memory to the `pipe_comm` pipe. The `pipe_comm` pipe is a pipe with fixed sized messages (`NU_FIXED_SIZE`), and each message will be 8 bits in size. The pipe is associated with the name “comm” and tasks that choose to suspend on this pipe will be resumed in First-In-First-Out order.

```
NU_Allocate_Memory(&dm_memory, &data_pipe, 32768, NU_NO_SUSPEND);
NU_Create_Pipe(&pipe_comm, "comm", data_pipe, 32768, NU_FIXED_SIZE,
              1, NU_FIFO);
}
```

The `entry_recv_1` and `entry_recv_2` functions serve as the entry point for the `task_recv_1` and `task_recv_2` tasks respectively. The tasks will continuously loop, issuing an `NU_Receive_From_Pipe` call for each iteration of the loop. The `NU_Receive_From_Pipe` will suspend until there is a message placed into the pipe (as indicated by `NU_SUSPEND`). Whenever a message is received, `NU_Receive_From_Pipe` will exit with a return value of `NU_SUCCESS`. After the call has returned, `recvmsg` will contain the message received. Therefore, there are two tasks that are continuously suspending on the same pipe, both waiting for a message to be placed into the pipe.

The PLUS scheduler will resume these tasks based on the `suspend_type` flag that was specified when the `pipe_comm` pipe was created.

```
void entry_recv_1(UNSIGNED argc, VOID *argv)
{
    CHAR recvmmsg;
    UNSIGNED actual_size;

    while(1)
    {
        if (NU_Receive_From_Pipe(&pipe_comm, &recvmmsg, 1, &actual_size,
                                NU_SUSPEND) == NU_SUCCESS)
        {
        }
        else
        {
        }
    }
}
```

```
void entry_recv_2(UNSIGNED argc, VOID *argv)
{
    CHAR recvmmsg;
    UNSIGNED actual_size;

    while(1)
    {
        if (NU_Receive_From_Pipe(&pipe_comm, &recvmmsg, 1, &actual_size,
                                NU_SUSPEND) == NU_SUCCESS)
        {
        }
        else
        {
        }
    }
}
```

The function `entry_send` serves as the task entry point for the `task_send` task. Note that the `task_recv_1` and `task_recv_2` tasks are of a higher priority, and will always be given first chance to run. Because of this, whenever `task_send` sends a message with `pipe_comm`, either `task_recv_1` or `task_recv_2` will be immediately resumed.

The `task_send` task continuously loops, and for each iteration of the loop it makes calls to two different PLUS services. The first service call is to `NU_Send_To_Pipe` which will send a single message with the `pipe_comm` pipe. The second service call that is issued is `NU_Broadcast_To_Pipe`, which will send the message to every task that is currently suspended on this pipe. Note that in this example, whenever this task is running, there will always be two tasks (`task_recv_1` and `task_recv_2`) suspended on the `pipe_comm` pipe. The result is that the message that is sent with `NU_Send_To_Pipe` will only be received by one of the suspended tasks, while the message sent with `NU_Broadcast_To_Pipe` will be received by both suspended tasks.



```
void entry_send(UNSIGNED argc, VOID *argv)
{
    UNSIGNED sendmsg;

    while(1)
    {
```

Assign decimal 1 to sendmsg, then issue NU_Send_To_Pipe on the pipe_comm pipe. Since two tasks will always be suspended on this pipe, and the pipe was created with the NU_FIFO suspension flag, the first task that suspended on the pipe will always receive this message.

```
        sendmsg=1;
        if (NU_Send_To_Pipe(&pipe_comm, &sendmsg, 1, NU_SUSPEND)
            == NU_SUCCESS)
        {
        }
        else
        {
        }
    }
```

Assign decimal 2 to sendmsg, then issue NU_Broadcast_To_Pipe on the pipe_comm pipe. Because the priority of task_recv_1 and task_recv_2 are of a higher priority than this task, we are guaranteed that two tasks will always be suspended on this pipe. Therefore, the result of the NU_Broadcast_To_Pipe service is that both tasks will be sent the message.

```
        sendmsg=2;
        if (NU_Broadcast_To_Pipe(&pipe_comm, &sendmsg, 1, NU_SUSPEND)
            == NU_SUCCESS)
        {
        }
        else
        {
        }
    }
}
```



Semaphores

Introduction

Function Reference

Example Source Code

Introduction

Semaphores provide a mechanism to control execution of critical sections of an application. Nucleus PLUS provides counting semaphores that range in value from 0 to 4,294,967,294. The two basic operations on a semaphore are *obtain* and *release*. Obtain-semaphore requests decrement the semaphore, while release-semaphore requests increment the semaphore.

Resource allocation is the most common application of a semaphore. Additionally, semaphores created with an initial value can be used to indicate an event.

Suspension

The obtain-semaphore service provides options for unconditional suspension, suspension with a timeout, and no suspension.

A task attempting to obtain a semaphore whose count is currently zero can suspend. Resumption of the task is possible when a release-semaphore request is made.

Multiple tasks may suspend trying to obtain a single semaphore. Tasks are suspended in either FIFO or priority order, depending on how the semaphore was created. If the semaphore supports FIFO suspension, tasks are resumed in the order in which they tried to obtain the semaphore. Otherwise, if the semaphore supports priority suspension, tasks are resumed from high priority to low priority.

Deadlock

A deadlock refers to a situation where two or more tasks are forever suspended attempting to obtain two or more semaphores. The simplest example of this situation is a system with two tasks and two semaphores. Suppose the first task has the second semaphore and the second task has the first semaphore. Now suppose that the second task attempts to obtain the second semaphore and the first task attempts to obtain the first semaphore. Since each task has the semaphore that the other needs, the tasks could suspend on the semaphores forever.

Prevention is the preferred way to deal with deadlocks. This technique imposes rules on how semaphores are used by the application. For example, if tasks are not allowed to possess more than one semaphore at a time, deadlocks are prevented. Alternatively, deadlocks may be prevented if tasks obtain multiple semaphores in the same order.

The optional timeout on obtain-semaphore suspension can be used to recover from a deadlock situation.

Priority Inversion

Priority inversion occurs when a higher priority task is suspended on a semaphore that a lower priority task has. This situation is unavoidable if different priority tasks share the same protected resources. In such situations, a limited and predictable amount of time in priority inversion is acceptable.



However, if the low priority task is preempted by a middle priority task during a priority inversion situation, the amount of time in priority inversion is no longer deterministic. Such a situation can be avoided by insuring that all tasks using the same semaphore have the same priority, at least while they own the semaphore.

Dynamic Creation

Nucleus PLUS semaphores are created and deleted dynamically. There is no preset limit on the number of semaphores an application may have. Each semaphore requires a control block. The memory for the control block is supplied by the application. Semaphores may be created with any initial count.

Determinism

Processing time required for obtaining and releasing semaphores is constant. However, the processing time required to suspend a task in priority order is affected by the number of tasks currently suspended on the semaphore.

Semaphore Information

Application tasks can obtain a list of active semaphores. Detailed information about each semaphore is also available. This information includes the semaphore name, current count, suspension type, number of tasks waiting, and the first task waiting.

Function Reference

The following function reference contains all functions related to Nucleus PLUS semaphores. The following functions are contained in this reference:

- NU_Create_Semaphore
- NU_Delete_Semaphore
- NU_Established_Semaphores
- NU_Obtain_Semaphore
- NU_Release_Semaphore
- NU_Reset_Semaphore
- NU_Semaphore_Information
- NU_Semaphore_Pointers



NU_Create_Semaphore

```
STATUS NU_Create_Semaphore(NU_SEMAPHORE *semaphore,
                           CHAR *name,
                           UNSIGNED initial_count,
                           OPTION suspend_type)
```

This service creates a counting semaphore. Semaphore values can range from 0 through 4,294,967,294.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Synchronization Services

Parameters

Parameter	Meaning
semaphore	Pointer to the user-supplied semaphore control block. Note: all subsequent requests made to the semaphore require this pointer.
name	Pointer to an 8 character name for the semaphore. The name does not have to be null-terminated.
initial_count	Specifies the initial count of the semaphore.
suspend_type	Specifies how tasks suspend on the semaphore. Valid options for this parameter are NU_FIFO and NU_PRIORITY, which represent First-In-First-Out (FIFO) and priority-order task suspension, respectively.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_SEMAPHORE	Indicates the semaphore control block pointer is NULL or is already in use.
NU_INVALID_SUSPEND	Indicates the suspend_type parameter is invalid.



Example

```

/* Assume semaphore control block "Semaphore" is defined
   as global data structure. This is one of several ways
   to allocate a control block. */

NU_SEMAPHORE      Semaphore;
.
.
/* Assume status is defined locally. */

STATUS  status;    /* Semaphore creation status */

/* Create a semaphore with an initial count of 1 and priority
   order task suspension. */

status =  NU_Create_Semaphore(&Semaphore, "any name", 1,
                              NU_PRIORITY);

/* status indicates if the service was successful. */

```

See Also

```

NU_Delete_Semaphore, NU_Established_Semaphores,
NU_Semaphore_Pointers, NU_Semaphore_Information

```



NU_Delete_Semaphore

```
STATUS NU_Delete_Semaphore(NU_SEMAPHORE *semaphore)
```

This service deletes a previously created semaphore. The parameter semaphore identifies the semaphore to delete. Tasks suspended on this semaphore are resumed with the appropriate error status. The application must prevent the use of this semaphore during and after deletion.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Synchronization Services

Parameters

Parameter	Meaning
semaphore	Pointer to the user-supplied semaphore control block.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_SEMAPHORE	Indicates the semaphore pointer is invalid.

Example

```
NU_SEMAPHORE      Semaphore;
STATUS             status
.
.
/* Delete the semaphore control block "Semaphore". Assume
   "Semaphore" has previously been created with the Nucleus
   PLUS NU_Create_Semaphore service call. */
status = NU_Delete_Semaphore(&Semaphore);

/* At this point, status indicates whether the service
   request was successful. */
```

See Also

NU_Create_Semaphore, NU_Established_Semaphores,
NU_Semaphore_Pointers, NU_Semaphore_Information



NU_Established_Semaphores

UNSIGNED NU_Established_Semaphores(VOID)

This service returns the number of established semaphores. All created semaphores are considered established. Deleted semaphores are no longer considered established.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Synchronization Services

Parameters

None

Return Value

This service call returns the number of created pipes in the system.

Example

```
UNSIGNED total_semaphores;

/* Obtain the total number of semaphores. */
total_semaphores = NU_Established_Semaphores( );
```

See Also

NU_Create_Semaphore, NU_Delete_Semaphore, NU_Semaphore_Pointers,
NU_Semaphore_Information

NU_Obtain_Semaphore

```
STATUS NU_Obtain_Semaphore(NU_SEMAPHORE *semaphore,
                           UNSIGNED suspend)
```

This service obtains an instance of the specified semaphore. Since “instances” are implemented with an internal counter, obtaining a semaphore translates into decrementing the semaphore’s internal counter by one. If the semaphore counter is zero before this call, the service cannot be immediately satisfied. The parameters of this service are further defined as follows:

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Synchronization Services

Parameters

Parameter	Meaning
semaphore	Pointer to the user-supplied semaphore control block.
suspend	Specifies whether or not to suspend the calling task if the semaphore cannot be obtained (is currently zero).

Suspension

The following table summarizes the possible values for the `suspend` parameter.

Suspension Option	Meaning
NU_NO_SUSPEND	The service returns immediately regardless of whether or not the request can be satisfied. Note: this is the only valid option if the service is called from a non-task thread.
NU_SUSPEND	The calling task is suspended until the message can be copied into the mailbox.
timeout value	(1 – 4,294,967,293). The calling task is suspended until the message can be copied into the mailbox or until the specified number of ticks has expired.



Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_SEMAPHORE	Indicates the semaphore pointer is invalid.
NU_INVALID_SUSPEND	Indicates that suspend attempted from a non-task thread.
NU_UNAVAILABLE	Indicates the semaphore is unavailable.
NU_TIMEOUT	Indicates that the semaphore is still unavailable even after suspending for the specified timeout value.
NU_SEMAPHORE_DELETED	Semaphore was deleted while the task was suspended.

Example

```

NU_SEMAPHORE    Semaphore;
STATUS          status
.
.
.
/* Obtain an instance of the semaphore control block
   "Semaphore". If the semaphore is unavailable, suspend
   for a maximum of 20 timer ticks. Note: the order of multiple
   tasks suspending on the same semaphore is determined when
   the semaphore is created. Assume "Semaphore" has previously
   been created with the Nucleus PLUS NU_Create_Semaphore
   service call. */
status = NU_Obtain_Semaphore(&Semaphore, 20);

/* At this point, status indicates whether the service request was
   successful. */

```

See Also

NU_Release_Semaphore, NU_Semaphore_Information

NU_Release_Semaphore

```
STATUS NU_Release_Semaphore(NU_SEMAPHORE *semaphore)
```

This service releases an instance of the semaphore specified by the parameter `semaphore`. If there are any tasks waiting to obtain the same semaphore, the first task waiting is given this instance of the semaphore. Otherwise, if there are no tasks waiting for this semaphore, the internal semaphore counter is incremented by one.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Synchronization Services

Parameters

Parameter	Meaning
<code>semaphore</code>	Pointer to the user-supplied semaphore control block.

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_SEMAPHORE</code>	Indicates the semaphore pointer is invalid.

Example

```
NU_SEMAPHORE    Semaphore;
STATUS          status;
.
.
.
/* Release an instance of the semaphore control block
"Semaphore". If other tasks are waiting to obtain the
same semaphore, this service results in a transfer of
this instance of the semaphore to the first task waiting.
Assume "Semaphore" has previously been created with the
Nucleus PLUS      NU_Create_Semaphore service call. */
status = NU_Release_Semaphore(&Semaphore);
```

See Also

`NU_Obtain_Semaphore`, `NU_Semaphore_Information`



NU_Reset_Semaphore

```
STATUS NU_Reset_Semaphore(NU_SEMAPHORE *semaphore,
                          UNSIGNED initial_count)
```

This service resets the semaphore specified by `semaphore` to the value of `initial_count`. All tasks suspended on the semaphore are resumed with the appropriate reset status.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Synchronization Services

Parameters

Parameter	Meaning
<code>semaphore</code>	Pointer to the user-supplied semaphore control block.
<code>initial_count</code>	Specifies the initial count of the semaphore.

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_SEMAPHORE</code>	Indicates the semaphore pointer is invalid.

Example

```
NU_SEMAPHORE    Semaphore;
STATUS          status
.
.
.
/* Reset the semaphore control block "Semaphore". The initial
count is set to 1. Assume "Semaphore" has previously been
created with the Nucleus PLUS NU_Create_Semaphore service call. */
status = NU_Reset_Semaphore(&Semaphore, 1);
```

See Also

`NU_Obtain_Semaphore`, `NU_Release_Semaphore`, `NU_Semaphore_Information`



NU_Semaphore_Information

```
STATUS NU_Semaphore_Information(NU_SEMAPHORE *semaphore,
                                CHAR *name,
                                UNSIGNED *current_count,
                                OPTION *suspend_type,
                                UNSIGNED *tasks_waiting,
                                NU_TASK **first_task)
```

This service returns various information about the specified task synchronization semaphore. The parameters of this service are further defined as follows:

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Synchronization Services

Parameters

Parameter	Meaning
semaphore	Pointer to the synchronization semaphore.
name	Pointer to an 8 character destination area for the semaphore's name.
current_count	Pointer to a variable to hold the current instance count of the semaphore.
suspend_type	Pointer to a variable that holds the task's suspend type. Valid task suspend types are NU_FIFO and NU_PRIORITY.
tasks_waiting	Pointer to a variable to hold the number of tasks waiting on the queue.
first_task	Pointer to a task pointer. The pointer of the first suspended task is placed in the task pointer.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_SEMAPHORE	Indicates the semaphore pointer is invalid.



Example

```

NU_SEMAPHORE    Semaphore;
CHAR            semaphore_name[8];
UNSIGNED        current_count;
OPTION          suspend_type;
UNSIGNED        tasks_suspended;
NU_TASK         *first_task;
STATUS          status;
.
.
.
/* Obtain information about the semaphore control block
"Semaphore". Assume "Semaphore" has previously been
created with the Nucleus PLUS NU_Create_Semaphore service
call. */
status = NU_Semaphore_Information(&Semaphore, semaphore_name,
                                &current_count,
                                &suspend_type,
                                &tasks_suspended,
                                &first_task);

/* If status is NU_SUCCESS, the other information is accurate. */

```

See Also

```

NU_Create_Semaphore, NU_Delete_Semaphore,
NU_Established_Semaphores, NU_Semaphore_Pointers

```

NU_Semaphore_Pointers

```
UNSIGNED NU_Semaphore_Pointers(NU_SEMAPHORE **pointer_list,
                                UNSIGNED maximum_pointers)
```

This service builds a sequential list of pointers to all established semaphores in the system.

Note: semaphores that have been deleted are no longer considered established. The parameter `pointer_list` points to the location for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Synchronization Services

Parameters

Parameter	Meaning
<code>pointer_list</code>	Pointer to an array of <code>NU_SEMAPHORE</code> pointers. This array will be filled with pointers of established semaphores in the system.
<code>maximum_pointers</code>	The maximum number of <code>NU_SEMAPHORE</code> pointers to place into the array. Typically this will be the size of the <code>pointer_list</code> array.

Return Value

This service call returns the number of created semaphores in the system.



Example

```
/* Define an array capable of holding 20 semaphore pointers. */
NU_SEMAPHORE      *Pointer_Array[20];
UNSIGNED          number;

/* Obtain a list of currently active semaphore pointers
   (Maximum of 20). */
number = NU_Semaphore_Pointers(&Pointer_Array[0], 20);

/* At this point, number contains the actual number
   of pointers in the list. */
```

See Also

```
NU_Create_Semaphore, NU_Delete_Semaphore,
NU_Established_Semaphores, NU_Semaphore_Information
```



Example Source Code

The following source code will demonstrate how to use the basic semaphore function calls. Semaphores are generally used to control access to either a mutually exclusive device, or to a piece of mutually exclusive data, such as a global variable. This example demonstrates both of these uses. The function `init_device()` demonstrates how a semaphore can be used to protect a global variable against being modified by multiple tasks simultaneously. To demonstrate using a semaphore to protect a mutually exclusive device, the function `write_to_device` uses the same semaphore as it's device protection mechanism.

Include all necessary Nucleus PLUS include files.

```
#include "nucleus.h"
```

Define a structure typedef called `BUFFER`, and declare an instance of this structure called `my_device`. This example will then protect this global variable by obtaining a semaphore before every modification.

```
typedef struct BUFFER_STRUCT
{
    CHAR buf[128];
    UNSIGNED read;
    UNSIGNED write;
    UNSIGNED num_entries;
} BUFFER;

BUFFER my_device;
```

This program will use four PLUS structures. The variables `task_1` and `task_2` are both task control blocks (`NU_TASK`) which will be used to write to the mutually exclusive device. The semaphore control block, `semaphore_device` will be used to control access to the device buffer. Lastly, the dynamic memory pool control block will be used to allocate any memory required by this program.

```
NU_TASK task_1;
NU_TASK task_2;
NU_SEMAPHORE semaphore_device;
NU_MEMORY_POOL dm_memory;
```

The two void pointers `stack_1`, `stack_2`, will each hold a pointer to a separate task stack. Although not demonstrated in this program, these pointers could be used at a later time in the program to deallocate the task stacks, or they could be discarded if the tasks stacks will never be deallocated.

```
VOID *stack_1;
VOID *stack_2;
```

Declare the task entry point function for each of the tasks. These will later be passed as a parameter to the `NU_Create_Task` call which will associate these functions with each of their respective tasks.



```
void entry_1(UNSIGNED argc, VOID *argv);
void entry_2(UNSIGNED argc, VOID *argv);
```

Two other functions will be used in this demonstration: `init_devices`, and `write_to_device`. The function `init_devices` will be used to initialize the global variable, and will be protected with the previously declared `semaphore_device`. The function `write_to_device` will use this same semaphore to protect the mutually exclusive device.

```
void init_device();
void write_to_device(CHAR writechar);
```

`Application_Initialize` is used to create any PLUS structures, allocate any required memory, and to perform any other system initialization that is necessary. Specific to this example, `Application_Initialize` is used to create the dynamic memory pool, `dm_memory`, allocate memory for, and create the two tasks: `task_1`, and `task_2`, and also create `semaphore_device`. Lastly, a call to the function `init_devices` is made to initialize the global structure `my_device`.

```
VOID Application_Initialize(VOID *first_available_memory)
{
```

Create the dynamic memory pool and associate it with the `dm_memory` control block. The memory pool will be 10240 bytes large, will start at `first_available_memory`, and, if memory is unavailable, tasks that choose to suspend will be resumed in First-In-First-Out order. The minimum allocation from this pool will be 128 bytes. For more information on the `NU_Create_Memory_Pool` call, or dynamic memory pools in general, see Chapter 4.

```
NU_Create_Memory_Pool(&dm_memory, "system", first_available_memory,
10240, 128, NU_FIFO);
```

For each task in the system, allocate 1024 bytes of memory for their respective stacks. With the `NU_Allocate_Memory` call, we are allocating a 1024 byte block of memory out of the `dm_memory` dynamic memory pool. A pointer to the newly allocated memory is assigned to `stack_1`, and `stack_2` respectively. The pointer to this memory allocation is passed to the `NU_Create_Task` call, which will use this memory as the task stack.

```
NU_Allocate_Memory(&dm_memory, &stack_1, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_1, "TASK1", entry_1, 0, NU_NULL, stack_1,
1024, 10, 2, NU_PREEMPT, NU_START);

NU_Allocate_Memory(&dm_memory, &stack_2, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_2, "TASK2", entry_2, 0, NU_NULL,
stack_2, 1024, 10, 2, NU_PREEMPT, NU_START);
```

Create the semaphore that will be used to protect the mutually exclusive structure. The `semaphore_device` semaphore is named “DEVICE”, is created with an initial count of 1, and tasks that choose to suspended on this semaphore will be resumed in First-In-First-Out order. Nucleus PLUS semaphores can be counting semaphores if the semaphore is created with a count higher than 1. In such a case, the semaphore can be obtained up to the number of times specified.

```
NU_Create_Semaphore(&semaphore_device, "DEVICE", 1, NU_FIFO);
```

Make the function call to `init_device`. This function will use the above created semaphore to protect the global structure `my_device`.

```
init_device();
}
```

Both tasks in the system (`task_1` and `task_2`) continuously loop, making a call to `write_to_device` for each iteration of the loop. In the case of `task_1` (which is associated to the entry point `entry_1`) the task writes a single character, “1”, to the device. Accordingly, `task_2` (associated to the entry point `entry_2`) will write a “2” to the device for each iteration of the loop.

```
void entry_1(UNSIGNED argc, VOID *argv)
{
    while(1)
    {
        write_to_device('1');
    }
}
```

```
void entry_2(UNSIGNED argc, VOID *argv)
{
    while(1)
    {
        write_to_device('2');
    }
}
```

The function `init_device` is used to simulate initializing a device. If using real hardware, this function may setup control registers, clear out data buffers, or any other device dependent initialization. In this example however, we will use a global structure, `my_device`, to simulate the device. Since this device is mutually exclusive it is protected by using the `semaphore_device` semaphore. Note that this protection is only necessary if multiple threads of execution could be initializing the device simultaneously.

```
void init_device()
{
```

Obtain the semaphore, `semaphore_device`. Since this semaphore was created with a count of 1, only one thread of execution can have possession of the semaphore at any given time. Therefore, we are guaranteed that only one task at a time can be modifying the `my_device` structure.

```
NU_Obtain_Semaphore(&semaphore_device, NU_SUSPEND);
```

Modify the global variable. In the case of real hardware, the following code could be replaced with control register initialization, clearing buffers, or any other device dependent initialization that may be required.



```
my_device.read = 0;
my_device.write = 0;
my_device.num_entries = 0;
```

When finished modifying the mutually exclusive data, release the semaphore so that other threads of execution can then modify the structure.

```
NU_Release_Semaphore(&semaphore_device);
}
```

Similar to the `init_device` function, the following function, `write_to_device` will use the `semaphore_device` semaphore to protect the mutually exclusive device. In this example, both `task_1`, and `task_2` (see their respective task entry points, `entry_1` and `entry_2`) are using this function to write to the device. Since the semaphore, `semaphore_device` was created as a binary semaphore (count 1), only one of these tasks can be modifying the device at any given time.

```
void write_to_device(CHAR writechar)
{
```

Make a call to `NU_Obtain_Semaphore` to obtain the semaphore. If a task already has possession of the semaphore, then the task making the second request will be suspended because suspension was requested by specifying the `NU_SUSPEND` option.

```
NU_Obtain_Semaphore(&semaphore_device, NU_SUSPEND);
```

Make any necessary modifications to the buffer. If actual hardware were being used, a transmit finished interrupt could be used to read data out of this buffer and place it onto the device. Alternately, one could choose not to use a buffer, and the following code could be replace with code to place the data onto the physical device.

```
my_device.buf[my_device.write] = writechar;
my_device.write++;

if (my_device.write >= 128)
my_device.write = 0;

if (my_device.num_entries < 128)
my_device.num_entries++;

else
my_device.read = my_device.write;
```

Release the `semaphore_device` semaphore so that other tasks can modify the device.

```
NU_Release_Semaphore(&semaphore_device);
}
```



10

Event Groups

Introduction

Function Reference

Example Source Code



Introduction

Event groups provide a mechanism to indicate that a certain system event has occurred. An event is represented by a single bit in an event group. This bit is called an event flag. There are 32 event flags in each event group.

Event flags can be set and cleared using logical AND/OR combinations. Event flags can be received in logical AND/OR combinations as well. Additionally, event flags may be reset automatically after they are received.

Suspension

The receive event flag requests provide options for unconditional suspension, suspension with a timeout, and no suspension.

A task attempting to receive a combination of event flags that are not present can suspend. Resumption of the task occurs when a set-event-flags operation satisfies the combination of events requested by the task.

Multiple tasks may suspend trying to receive different combinations of event flags from the same event group. All tasks suspended on an event group are checked for resumption when a set-event-flags operation is performed on the event group.

Dynamic Creation

Nucleus PLUS event groups are created and deleted dynamically. There is no preset limit on the number of event groups an application may have. Each event group requires a control block. The memory for the control block is supplied by the application.

Determinism

Processing time required for receiving event flags from an event group is constant. However, the processing time required to set event flags in an event group is affected by the number of tasks suspended on the event group.

Event Group Information

Application tasks may obtain a list of active event groups. Detailed information about each event group is also available. This information includes the event group name, current event flags, number of tasks waiting, and the first task waiting.



Function Reference

The following function reference contains all functions related to Nucleus PLUS event groups. The following functions are contained in this reference:

- NU_Create_Event_Group
- NU_Delete_Event_Group
- NU_Established_Event_Groups
- NU_Event_Group_Information
- NU_Event_Group_Pointers
- NU_Retrieve_Events
- NU_Set_Events



NU_Create_Event_Group

```
STATUS NU_Create_Event_Group(NU_EVENT_GROUP *group,
                             CHAR *name)
```

This service creates an event flag group. Each event flag group contains 32 event flags. All event flags are initially set to 0. The parameters to this service are further defined as follows:

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Synchronization Services

Parameters

Parameter	Meaning
group	Pointer to the user-supplied event flag group control block. Note: all subsequent requests made to the event group require this pointer.
name	Pointer to an 8-character name for the event flag group. The name does not have to be null-terminated.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_GROUP	Indicates the event group control block pointer is NULL or is already in use.



Example

```

/* Assume event group control block "Events" is defined
   as a global data structure. This is one of several ways
   to allocate a control block. */

NU_EVENT_GROUP  Events;
.
.
/* Assume status is defined locally. */
STATUS          status; /* Event group creation status */

/* Create an event flag group. */
status = NU_Create_Event_Group(&Events, "any name");

/* At this point status indicates if the service was successful. */

```

See Also

```

NU_Delete_Event_Group, NU_Established_Event_Groups,
NU_Event_Group_Pointers, NU_Event_Group_Information

```

NU_Delete_Event_Group

```
STATUS NU_Delete_Event_Group(NU_EVENT_GROUP *group)
```

This service deletes a previously created event flag group. The parameter `group` identifies the event flag group to delete. Tasks suspended on this event group are resumed with the appropriate error status. The application must prevent the use of this event group during and after deletion.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Synchronization Services

Parameters

Parameter	Meaning
<code>group</code>	Pointer to the user-supplied event flag group control block.

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_GROUP</code>	Indicates the event flag group pointer is invalid.

Example

```
NU_EVENT_GROUP    Group;
STATUS            status
.
.
/* Delete the event flag group control block "Group".
   Assume "Group" has previously been created with
   the Nucleus PLUS NU_Create_Event_Group service call. */
status = NU_Delete_Event_Group(&Group);

/* At this point, status indicates whether the service
   request was successful. */
```

See Also

`NU_Create_Event_Group`, `NU_Established_Event_Groups`,
`NU_Event_Group_Pointers`, `NU_Event_Group_Information`



NU_Established_Event_Groups

UNSIGNED NU_Established_Event_Groups(VOID)

This service returns the number of established event-flag groups. All created event-flag groups are considered established. Deleted event-flag groups are no longer considered established.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Synchronization Services

Parameters

None

Return Value

This service call returns the number of created event groups in the system.

Example

```
UNSIGNED total_event_groups;

/* Obtain the total number of event flag groups. */
total_event_groups = NU_Established_Event_Groups( );
```

See Also

NU_Create_Event_Group, NU_Delete_Event_Group,
NU_Event_Group_Pointers, NU_Event_Group_Information

NU_Event_Group_Information

```
STATUS NU_Event_Group_Information(NU_EVENT_GROUP *group,
                                CHAR *name,
                                UNSIGNED *event_flags,
                                UNSIGNED *tasks_waiting,
                                NU_TASK **first_task)
```

This service returns various information about the specified event flag group.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Synchronization Services

Parameters

Parameter	Meaning
group	Pointer to the user-supplied event flag group control block.
name	Pointer to an 8-character destination area for the event flag group's name.
event_flags	Pointer to a variable to hold the current event flags.
tasks_waiting	Pointer to a variable to hold the number of tasks waiting on the event flag group.
first_task	Pointer to a task pointer. The pointer of the first suspended task is placed in this task pointer.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_GROUP	Indicates the event flag group pointer is invalid.

Example

```

NU_EVENT_GROUP    Group;
CHAR              group_name[8];
UNSIGNED          event_flags;
UNSIGNED          tasks_suspended;
NU_TASK           *first_task;
STATUS            status
.
.
.
/* Obtain information about the event group control block
   "Group". Assume "Group" has previously been created
   with the Nucleus PLUS NU_Create_Event_Group service call. */
status = NU_Event_Group_Information(&Group, group_name,
                                   &event_flags,
                                   &tasks_suspended,
                                   &first_task);

/* If status is NU_SUCCESS, the other information is
   accurate. */

```

See Also

NU_Create_Event_Group, NU_Delete_Event_Group,
 NU_Established_Event_Groups, NU_Event_Group_Pointers



NU_Event_Group_Pointers

```
UNSIGNED NU_Event_Group_Pointers(NU_EVENT_GROUP *pointer_list,
                                UNSIGNED maximum_pointers)
```

This service builds a sequential list of pointers to all established event-flag groups in the system. **Note:** event flag-groups that have been deleted are no longer considered established. The parameter `pointer_list` points to the location for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Synchronization Services

Parameters

Parameter	Meaning
<code>pointer_list</code>	Pointer to an array of <code>NU_EVENT_GROUP</code> pointers. This array will be filled with pointers of established semaphores in the system.
<code>maximum_pointers</code>	The maximum number of <code>NU_EVENT_GROUP</code> pointers to place into the array. Typically this will be the size of the <code>pointer_list</code> array.

Return Value

This service call returns the number of created event groups in the system.

Example

```

/* Define an array capable of holding 20 event flag
   group pointers. */
NU_EVENT_GROUP    *Pointer_Array[20];
UNSIGNED          number;

/* Obtain a list of currently active event flag group
   pointers (Maximum of 20). */
number = NU_Event_Group_Pointers(&Pointer_Array[0], 20);

/* At this point, number contains the actual number
   of pointers in the list. */

```

See Also

```

NU_Create_Event_Group, NU_Delete_Event_Group,
NU_Established_Event_Groups, NU_Event_Group_Information

```

NU_Retrieve_Events

```
STATUS NU_Retrieve_Events(NU_EVENT_GROUP *group,
                          UNSIGNED requested_events,
                          OPTION operation,
                          UNSIGNED *retrieved_events,
                          UNSIGNED suspend)
```

This service retrieves the specified event-flag combination from the specified event-flag group. If the combination is present, the service completes immediately.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Synchronization Services

Parameters

Parameter	Meaning
group	Pointer to the user-supplied event flag group control block.
requested_events	Requested event flags. A set bit indicates the corresponding event flag is requested.
operation	There are four operation options available: NU_AND, NU_AND_CONSUME, NU_OR, and NU_OR_CONSUME. NU_AND and NU_AND_CONSUME options indicate that all of the requested event flags are required. NU_OR and NU_OR_CONSUME options indicate that one or more of the requested event flags is sufficient. The CONSUME option automatically clears the event flags present on a successful request.
retrieved_events	Contains the event flags actually retrieved.
suspend	Specifies whether or not to suspend the calling task if the requested event flag combination is not available.

Suspension

The following table summarizes the possible values for the `suspend` parameter.

Suspension Option	Meaning
<code>NU_NO_SUSPEND</code>	The service returns immediately regardless of whether or not the request can be satisfied. Note: this is the only valid option if the service is called from a non-task thread.
<code>NU_SUSPEND</code>	The calling task is suspended until the message can be copied into the mailbox.
<code>timeout value</code>	(1 – 4,294,967,293). The calling task is suspended until the message can be copied into the mailbox or until the specified number of ticks has expired.

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_GROUP</code>	Indicates the event flag group pointer is invalid.
<code>NU_INVALID_POINTER</code>	Indicates the retrieved event flag pointer is <code>NULL</code> .
<code>NU_INVALID_OPERATION</code>	Indicates the <code>operation</code> parameter is invalid.
<code>NU_INVALID_SUSPEND</code>	Indicates that <code>suspend</code> attempted from a non-task thread.
<code>NU_NOT_PRESENT</code>	Indicates the requested event flag combination is not currently present.
<code>NU_TIMEOUT</code>	Indicates the requested event flag combination is not present even after the specified suspension timeout.
<code>NU_GROUP_DELETED</code>	Indicates the event flag group was deleted while the task was suspended.

Example

```
NU_EVENT_GROUP    Group;
UNSIGNED          actual_flags;
STATUS            status;
.
.
.
/* Retrieve event flags 7, 2, and 1 from the event group
   control block "Group". Note: all event flags must be
   present to satisfy the request. If they are not, the
   calling task suspends unconditionally. Also, event
   flags 7, 2, and 1 are consumed when this request is
   satisfied. Assume "Group" has previously been created
   with the Nucleus PLUS NU_Create_Event_Group service call. */
status = NU_Retrieve_Events(&Group, 0x86, NU_AND_CONSUME,
                           &actual_flags, NU_SUSPEND);
```

See Also

NU_Set_Events, NU_Event_Group_Information



NU_Set_Events

```
STATUS NU_Set_Events(NU_EVENT_GROUP *group,
                    UNSIGNED event_flags,
                    OPTION operation)
```

This service sets the specified event flags in the specified event group. Any task waiting on the event group whose event flag request is satisfied by this service is resumed.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Synchronization Services

Parameters

Parameter	Meaning
group	Pointer to the user-supplied event flag group control block.
event_flags	Event flag values.
operation	There are two operation options available: NU_OR and NU_AND. NU_OR causes the event flags specified to be "ORed" with the current event flags in the group. NU_AND causes the event flags specified to be "ANDed" with the current event flags in the group. Note: event flags can be cleared with the NU_AND option.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_GROUP	Indicates the event flag group pointer is invalid.
NU_INVALID_OPERATION	Indicates the operation parameter is invalid.

Example

```
NU_EVENT_GROUP    Group;
STATUS            status;
.
.
.
/* Set event flags 7, 2, and 1 in the event group control
   block "Group". Assume "Group" has previously been
   created with the Nucleus PLUS NU_Create_Event_Group service
   call.*/
status = NU_Set_Events(&Group, 0x00000086 NU_OR);

/* If status is NU_SUCCESS the event flags were set. */
```

See Also

NU_Retrieve_Events, NU_Event_Group_Information



Sample Source Code

Include all necessary Nucleus PLUS include files.

```
#include "nucleus.h"
```

There are two possible events, which will be represented by the definitions of EVENT_1 and EVENT_2. The #define of WAIT_EVENTS will be used by the NU_Retrieve_Events function call to suspend on both individual events.

```
#define EVENT_1      0x00000001
#define EVENT_2      0x00000002
#define WAIT_EVENTS  0x00000003
```

We will use five different Nucleus PLUS structures in this example program. All necessary memory will be allocated out of the dynamic memory pool, dm_memory. There are also three NU_TASK structures which will be used for the three tasks in the system. One task (task_wait) will be of a higher priority, and will suspend on the NU_EVENT_GROUP, eg_wait. The remaining two tasks, task_set1, and task_set2 will set the above defined events, EVENT_1 and EVENT_2. When both of these bits are set task_wait will be resumed.

```
NU_MEMORY_POOL dm_memory;
NU_TASK task_wait;
NU_TASK task_set1;
NU_TASK task_set2;
NU_EVENT_GROUP eg_wait;
```

Three void pointers will be used in this example. Each void pointer will hold a pointer to a separate task stack. Although not demonstrated in this program, these pointers could be used at a later time in the program to deallocate the task stacks, or they could be discarded if the task stacks will never be deallocated.

```
VOID *stack_wait;
VOID *stack_set1;
VOID *stack_set2;
```

Declare the task entry point function for each of the three tasks. These will later be passed as a parameter to the NU_Create_Task call which will associate these functions with each of their respective tasks.

```
void wait(UNSIGNED argc, VOID *argv);
void set1(UNSIGNED argc, VOID *argv);
void set2(UNSIGNED argc, VOID *argv);
```

Application_Initialize will be used to create the dynamic memory pool, out of which memory will be allocated for the three tasks in the system. Application_Initialize will also be used to create the event group, create the three tasks, and associate each of the tasks with a newly created task stack.

```
VOID Application_Initialize(VOID *first_available_memory)
{
```



Create the dynamic memory pool, and associate it with the `dm_memory` control block. The memory pool will be 10240 bytes large, will start at `first_available_memory`, and, if memory is unavailable, tasks that choose to suspend on this memory pool will be resumed in First-In-First-Out order. The minimum allocation from this pool will be 128 bytes. For more information on the `NU_Create_Memory_Pool` call, or dynamic memory pools in general, see Chapter 4.

```
NU_Create_Memory_Pool(&dm_memory, "sysmem", first_available_memory,
                    10240, 128, NU_FIFO);
```

For each task in the system, allocate 1024 bytes of memory for their respective stacks. With the `NU_Allocate_Memory` call, we are allocating a 1024 byte block of memory out of the `dm_memory` dynamic memory pool. A pointer to the newly allocated memory is assigned to `stack_wait`, `stack_set1`, and `stack_set2` respectively. The pointer to this memory allocation is passed to the `NU_Create_Task` call, which will use this memory as its task stack.

```
NU_Allocate_Memory(&dm_memory, &stack_wait, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_wait, "WAIT", wait, 0, NU_NULL, stack_wait,
              1024, 3, 0, NU_PREEMPT, NU_START);

NU_Allocate_Memory(&dm_memory, &stack_set1, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_set1, "SET1", set1, 0, NU_NULL, stack_set1,
              1024, 4, 0, NU_PREEMPT, NU_START);

NU_Allocate_Memory(&dm_memory, &stack_set2, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_set2, "SET2", set2, 0, NU_NULL, stack_set2,
              1024, 4, 0, NU_PREEMPT, NU_START);
```

Use `NU_Create_Event_Group` to create an event group with the text name of “WAIT.” The tasks `task_wait`, `task_set1`, and `task_set2` will use this event group to synchronize their activity.

```
NU_Create_Event_Group(&eg_wait, "WAIT");
}
```

The function `wait` is the entry point for the `task_wait` task. The `task_wait` task will suspend on the `eg_wait` event group until both `EVENT_1` and `EVENT_2` are set by the `task_set1` and `task_set2` tasks.

```
void wait(UNSIGNED argc, VOID *argv)
{
```

The variable `retrieved` will be passed as a parameter to `NU_Retrieve_Events`. Upon successful completion of that service call, it will contain the events that were actually retrieved. The value of this variable can then be used in a construct such as a case statement to perform different actions based upon which signal (represented by distinct bit patterns) was actually sent.

```
UNSIGNED retrieved;
```



Use the `NU_Retrieve_Events` service call to suspend until all events are set. Since this is the highest priority task in the system (see the `NU_Create_Task` service calls in the `Application_Initialize` function) it will be run first. Therefore, the `task_wait` task will suspend until the bits specified in `WAIT_EVENTS` are set.

The `NU_Retrieve_Events` service call will suspend, the result of the `NU_SUSPEND` parameter, on the `eg_wait` event group waiting for all bits in `WAIT_EVENTS` to be set. This behavior could also be modified by changing the `NU_AND` parameter to `NU_OR`, which would cause the `NU_Retrieve_Events` service call to suspend until any of the specified events were set. Consuming, or clearing, of event bits is also available by using the `NU_AND_CONSUME`, and `NU_OR_CONSUME` options.

```
if (NU_Retrieve_Events(&eg_wait, WAIT_EVENTS, NU_AND, &retrieved,
NU_SUSPEND) == NU_SUCCESS)
{
/* The requested events were successfully retrieved. */
}
}
```

The `task_set1` and `task_set2` tasks will both set a separate bit in the `eg_wait` event group. When these tasks are run, `task_wait` has already run and has suspended on the `eg_wait` event group. Since these two tasks are the only two remaining tasks in the system, and are of the same priority, they will be run consecutively and will each set their respective bits. After the second `NU_Set_Events` call is executed, `task_wait` will be immediately resumed to continue processing.

```
void set1(UNSIGNED argc, VOID *argv)
{
    NU_Set_Events(&eg_wait, EVENT_1, NU_OR);
}

void set2(UNSIGNED argc, VOID *argv)
{
    NU_Set_Events(&eg_wait, EVENT_2, NU_OR);
}
```



11

Signals

Introduction

Function Reference

Example Source Code



Introduction

Signals are in some ways similar to event flags. However, there are significant differences in operation. Event flag usage is synchronous by nature. The task does not recognize event flags are present until the specific service request is made. Signals operate in an asynchronous manner. When a signal is present, a special signal handling routine, previously designated by the task, is executed when the task is resumed. Each task is capable of handling 32 signals. Each signal is represented by a single bit.

Signal Handling Routine

The task's signal-handling routine must be supplied before any signals are processed. Processing inside a signal-handling routine has virtually the same constraints as a high-level interrupt service routine. Basically, most Nucleus PLUS services are available, provided self-suspension is avoided.

Enable Signal Handling

By default, tasks are created with all signals disabled. Individual signals may be enabled and disabled dynamically by each task.

Clearing Signals

Signals are automatically cleared when signal handling is invoked. Additionally, signals are cleared when a solicited request to receive signals is made. **Note:** tasks cannot suspend on solicited requests to receive signals.

Multiple Signals

Signals for a task are cleared once the signal-handling routine is started. Signal-handling routines are not interrupted by new signals. Processing of any new signals takes place after the current signal-processing completes. Identical signals sent before the first signal is recognized are discarded.

Determinism

Processing time required to send and receive signals is constant, at least in the worst case. Of course the time required to execute a signal-handling routine is application specific.



Function Reference

The following function reference contains all functions related to Nucleus PLUS signals. The following functions are contained in this reference:

```

NU_Control_Signals
NU_Receive_Signals
NU_Register_Signal_Handler
NU_Send_Signals

```

NU_Control_Signals

```
UNSIGNED NU_Control_Signals(UNSIGNED enable_signal_mask)
```

This service enables and/or disables signals of the calling task. There are 32 signals available for each task. Each signal is represented by a bit in `signal_enable_mask`. Signal 0 is represented by bit 0 and signal 31 is represented by bit 31. Setting a bit in `signal_enable_mask` enables the corresponding signal, while clearing a bit disables the corresponding signal. **Note:** the signal enable mask is cleared during task creation.

Overview

Option	
Tasking Changes	No
Allowed From	Task
Category	Task Synchronization Services

Parameters

Parameter	Meaning
<code>enable_signal_mask</code>	Bit pattern representing valid signals.

Return Value

This service returns the previous signal enable/disable mask.

Example

```

UNSIGNED old_signal_mask; /* Previous signal mask */
/* Lockout all of the current task's signals temporarily. */
old_signal_mask = NU_Control_Signals(0);
.
.
.
/* Restore previous signal mask. */
NU_Control_Signals(old_signal_mask);

```

See Also

`NU_Send_Signals`, `NU_Receive_Signals`, `NU_Register_Signal_Handler`



NU_Receive_Signals

UNSIGNED NU_Receive_Signals(VOID)

This service returns the current value of each signal associated with the calling task. All signals are automatically cleared as a result of the service call.

Overview

Option	
Tasking Changes	No
Allowed From	Task
Category	Task Synchronization Services

Parameters

None

Return Value

This service call returns the current value of each signal associated with the calling task.

Example

```
UNSIGNED signals;  
/* Receive and clear the signals of the current task. */  
signals = NU_Receive_Signals( );
```

See Also

NU_Control_Signals, NU_Register_Signal_Handler, NU_Send_Signals



NU_Register_Signal_Handler

```
STATUS NU_Register_Signal_Handler(VOID(*signal_handler)(UNSIGNED))
```

This service registers a signal handler pointed to by `signal_handler`, for the calling task. By default, all signals are disabled when the task is created. Signals remain disabled, regardless of `NU_Control_Signals` service requests, until a signal handler is registered for the task. A signal handler executes on top of the task's context. Most services can be called from a signal handler. However, services called from a signal handler cannot specify suspension.

Overview

Option	
Tasking Changes	No
Allowed From	Task
Category	Task Synchronization Services

Parameters

Parameter	Meaning
<code>signal_handler</code>	Function to be called whenever valid signals are received.

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_TASK</code>	Indicates the supplied task pointer is invalid.
<code>NU_INVALID_POINTER</code>	Indicates the signal handler pointer is <code>NULL</code> .



Example

```
STATUS  status;

/* Register the function "Signal_Handler" as the task's
   signal handler. */

void  Signal_Handler(UNSIGNED signals)
{
    /* Process relative to the singls present. Note that
       processing has the same constraints has HISRs in
       that self-suspension is not permitted. */
}

status = NU_Register_Signal_Handler(Signal_Handler);

/* If status is NU_SUCCESS, Signal_Handler is invoked
   each time enabled signals are sent. */
```

See Also

NU_Control_Signals, NU_Receive_Signals, NU_Send_Signals



NU_Send_Signals

```
STATUS NU_Send_Signals(NU_TASK *task, UNSIGNED signals)
```

This service sends the signals indicated by the parameter `signals` to the task pointed to by the parameter `task`. If the receiving task has any of the designated signals enabled, its registered signal handler is executed as soon as the receiving task's priority permits. Each task has 32 available signals that are represented by each bit in `signals`.

There are several conditions that prevent the receiving task's signal handler from being executed, as follows:

- Receiving task is in a finished or terminated state.
- Receiving task is unconditionally suspended (either it was not started after creation or it was suspended by `NU_Suspend_Task`). If this is the case, the signal handler does not execute until the task is resumed.
- There is always a task ready at a higher priority than the receiving task.
- The receiving task has not enabled the signals sent.
- The receiving task has not registered a signal handler.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Task Synchronization Services

Parameters

Parameter	Meaning
<code>task</code>	Pointer to the user-supplied task control block.
<code>signals</code>	Bit pattern representing signals to be sent.

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_TASK</code>	Indicates the task pointer is invalid.



Example

```
NU_TASK      Task;
STATUS       status;
.
.
.
/* Send signals 1, 7, and 31 to the task control block
   "Task". Notice that the signals correspond to the bit
   position. Assume "Task" has previously been created
   with the Nucleus PLUS NU_Create_Task service call. */
status = NU_Send_Signals(&Task, 0x80000082);
```

See Also

NU_Receive_Signals, NU_Control_Signals, NU_Register_Signal_Handler



Sample Source Code

In this example we will look at how Nucleus PLUS signals could be used to implement a control task which will perform various system tasks. Specific to this example, a task will signal that it can now be deleted, and removed from the system.

Include all necessary Nucleus PLUS include files.

```
#include "nucleus.h"
```

In this demonstration, a task, specifically `task_1` will send a signal to the control task, `task_control`, when it has finished processing. The `task_control` task will then delete the task, and deallocate the memory used for its stack. The `#define TASK_1_FINISHED` will be used to represent this signal.

```
#define TASK_1_FINISHED    0x00000001
```

Three Nucleus PLUS structures will be used in this example. The dynamic memory pool control block, `dm_memory`, will be used as the memory pool out of which all memory will be allocated. The two `NU_TASK` structures are the task control blocks for the two tasks in the system: `task_control`, and `task_1`.

```
NU_MEMORY_POOL dm_memory;
NU_TASK task_control;
NU_TASK task_1;
```

Two void pointers will be used in this example. Each void pointer will hold a pointer to a separate task stack. The void pointer, `stack_task_1` will be used in this example to deallocate the memory associated with the `task_1` stack.

```
VOID *stack_control;
VOID *stack_task_1;
```

Declare the task entry point function for each of the three tasks. These will later be passed as a parameter to the `NU_Create_Task` call which will associate these functions with each of their respective tasks.

```
void control(UNSIGNED argc, VOID *argv);
void entry_1(UNSIGNED argc, VOID *argv);
void sh_control(UNSIGNED signals);
```

`Application_Initialize` will be used to create the dynamic memory pool, out of which memory will be allocated for the two tasks in the system, which will then be created with the `NU_Create_Task` service call. After `Application_Initialize` executes, all tasks will be created, and the system will be ready to begin executing in a multi-tasking environment.

```
VOID Application_Initialize(VOID *first_available_memory)
{
```



Create the dynamic memory pool, and associate it with the `dm_memory` control block. The memory pool will be 10240 bytes large, will start at `first_available_memory`, and, if memory is unavailable, tasks that choose to suspend will be resumed in First-In-First-Out order. The minimum allocation from this pool will be 256 bytes. For more information on the `NU_Create_Memory_Pool` call, or dynamic memory pools in general, see Chapter 4.

```
NU_Create_Memory_Pool(&dm_memory, "system", first_available_memory,
                     10240, 256, NU_FIFO);
```

For each task in the system, allocate 1024 bytes of memory for their respective stacks. With the `NU_Allocate_Memory` call, we are allocating a 1024 byte block of memory out of the `dm_memory` dynamic memory pool. A pointer to the newly allocated memory is assigned to `stack_control`, and `stack_task_1` respectively. The pointer to this memory allocation is passed to the `NU_Create_Task` call, which will use this memory as the task stack.

```
NU_Allocate_Memory(&dm_memory, &stack_control, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_control, "CONTROL", control, 0, NU_NULL,
              stack_control, 1024, 11, 0, NU_PREEMPT, NU_START);

NU_Allocate_Memory(&dm_memory, &stack_task_1, 1024, NU_NO_SUSPEND);
NU_Create_Task(&task_1, "TASK1", entry_1, 0, NU_NULL, stack_task_1,
              1024, 10, 0, NU_PREEMPT, NU_START);
}
```

In order for a Nucleus PLUS task to receive signals, first, a signal handler must be associated with that task. The `task_control` task is used to register and control the `sh_control` signal handler. In a complete system, this task could also be used to run periodic system maintenance that did not depend on a signal being issued.

```
void control(UNSIGNED argc, VOID *argv)
{
```

The `NU_Register_Signal_Handler` service call associates a signal handling function with a specific task. After this call, upon a signal being sent to this task, the associated signal handler function will be executed. The associated function will be responsible for determining which signal was sent, and to take the correct action.

```
    NU_Register_Signal_Handler(&sh_control);
```

The task also needs to be informed of which signals in the system it should respond to. The `NU_Control_Signals` service call will set the required flags so that the signal handler function is only executed when valid signals are sent.

```
    NU_Control_Signals(TASK_1_FINISHED);
```

For this demonstration, if this task is ever executed, sleep for 10 timer ticks. In a real system, code could be inserted to do periodic maintenance regardless of whether a signal was sent to this task.

```
while(1)
{
    NU_Sleep(10);
}
}
```



The `sh_control` function is the signal handler that was associated with the `task_control` task. It is responsible for examining the current set of signals, evaluating what action to take, and then executing the correct code to handle that particular signal (or set of signals). Specific to this example, the signal handler will determine if `TASK_1_FINISHED` was sent, and if so, delete the task, and deallocate the memory used for its task stack.

```
void sh_control(UNSIGNED signals)
{
```

First determine if `TASK_1_FINISHED` was actually sent to the control task. If `TASK_1_FINISHED` was sent, then delete the task with a call to `NU_Delete_Task`, and deallocate the memory for the task's stack with a call to `NU_Deallocate_Memory`.

```
if (signals & TASK_1_FINISHED)
{
    NU_Delete_Task(&task_1);
    NU_Deallocate_Memory(&stack_task_1);
}
```

Use `NU_Receive_Signals` to clear the current set of signals.

```
NU_Receive_Signals();
}
```

In this demonstration, `task_1` is used to send a signal to the control task indicating that it has completed processing, and can now be removed from the system. Therefore, `entry_1`, the entry point for `task_1`, issues a call with `NU_Send_Signals` to send the `TASK_1_FINISHED` signal to `task_control`.

```
void entry_1(UNSIGNED argc, VOID *argv)
{
    NU_Send_Signals(&task_control, TASK_1_FINISHED);
}
```



12

Timers

Introduction

Function Reference

Example Source Code



Introduction

Most real-time applications require processing on periodic intervals of time. Each Nucleus PLUS task has a built-in timer. This timer is used to provide task sleeping and service call suspension timeouts.

Ticks

A tick is the basic unit of time for all Nucleus PLUS timer facilities. Each tick corresponds to a single hardware timer interrupt. The amount of actual time a tick represents is usually user-programmable.

Margin of Error

A timer request may be satisfied as much as one tick early in actual time. This is because a tick can occur immediately after the timer request. Therefore, the first tick of a timer request represents an actual time ranging from zero to the rate of the hardware timer interrupt. For example, the amount of actual time expired for a request of n ticks falls between the actual time n and $n-1$ ticks represent.

Hardware Requirement

Nucleus PLUS timer services require a periodic timer interrupt from the hardware. Without such an interrupt, timer facilities will not function. However, other Nucleus PLUS facilities are not affected by the absence of timer facilities.

Continuous Clock

Nucleus PLUS maintains a continuous counting tick clock. The maximum value of this clock is 4,294,967,294. The clock automatically resets on the tick after the maximum value is reached.

This continuous clock is reserved exclusively for application use. It may be read from and written to by the application at any time.

Task Timers

Each task has a built-in timer. This timer is used for task-sleep requests and suspension timeout requests. Additionally, a time-slice timer is available for tasks that require time-slicing.



Application Timers

Nucleus PLUS provides programmable timers for applications. These timers execute a specific user-supplied routine when they expire. The user-supplied expiration routine executes as a high-level interrupt service routine. Therefore, self-suspension requests are not allowed. Additionally, processing should be kept to a minimum.

Re-Scheduling

When a timer expires, the prescribed expiration routine is executed. After execution is complete, the timer is either dormant or rescheduled. If the timer's reschedule value is zero, it is dormant after the initial expiration. However, if the timer's reschedule value is nonzero, it is rescheduled to expire at that interval.

Enable/Disable

Application timers may be automatically enabled during creation. Additionally, timers may be enabled and disabled dynamically.

Reset

The initial ticks, rescheduling rate, and the expiration routine of a timer may be reset dynamically by the application.

Dynamic Creation

Nucleus PLUS application timers are created and deleted dynamically. There is no preset limit on the number of timers an application may have. Each timer requires a control block. The memory for this is supplied by the application.

Determinism

Processing time required to create, enable, disable, and modify application timers is constant. However, processing time required to execute the user-supplied expiration routines depends on the expiration routines themselves and the number of timers that expire simultaneously.



Timer Information

Application tasks may obtain a list of active timers. Detailed information about each timer is also available. This information includes the timer name, status, initial ticks, reschedule value, remaining ticks, and the expiration count.

Function Reference

The following function reference contains all functions related to Nucleus PLUS timers. The following functions are contained in this reference:

- NU_Control_Timer
- NU_Create_Timer
- NU_Delete_Timer
- NU_Established_Timers
- NU_Reset_Timer
- NU_Retrieve_Clock
- NU_Set_Clock
- NU_Timer_Information
- NU_Timer_Pointers



NU_Control_Timer

STATUS NU_Control_Timer(NU_TIMER *timer, OPTION enable)

This service enables or disables the application timer pointed to by `timer`. Legal values for the `enable` parameter are `NU_ENABLE_TIMER` and `NU_DISABLE_TIMER`.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, task
Category	Timer Services

Parameters

Parameter	Meaning
<code>timer</code>	Pointer to the user-supplied timer control block.
<code>enable</code>	Valid options for this parameter are <code>NU_ENABLE_TIMER</code> and <code>NU_DISABLE_TIMER</code> . <code>NU_ENABLE_TIMER</code> immediately after the function call. <code>NU_DISABLE_TIMER</code> leaves the timer disabled.

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_TIMER</code>	Indicates the timer pointer is invalid.
<code>NU_INVALID_ENABLE</code>	Indicates the enable parameter is invalid.

Example

```

NU_TIMER    Timer;
STATUS      status
.
.
.
/* Disable the timer control block "Timer". Assume
   "Timer" has previously been created with the
   Nucleus PLUS NU_Create_Timer service call. */

status =  NU_Control_Timer(&Timer, NU_DISABLE_TIMER);

/* At this point, status can be examined to determine
   whether the service request was successful. */

```

See Also

`NU_Create_Timer`, `NU_Reset_Timer`, `NU_Timer_Information`



NU_Create_Timer

```
STATUS NU_Create_Timer(NU_TIMER *timer, CHAR *name,
                      VOID (*expiration_routine)(UNSIGNED),
                      UNSIGNED id, UNSIGNED initial_time,
                      UNSIGNED reschedule_time,
                      OPTION enable)
```

This service creates an application timer. The specified expiration routine is executed each time the timer expires. Application expiration routines should avoid task suspension options. Suspension of the expiration routine can cause delays in other application timer requests.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, task
Category	Timer Services

Parameters

Parameter	Meaning
timer	Pointer to the user-supplied timer control block. Note: all subsequent requests made to the timer require this pointer.
name	Pointer to an 8 character name for the timer. The name does not have to be null-terminated.
expiration_routine	Specifies the application routine to execute when the timer expires.
id	An UNSIGNED data element supplied to the expiration routine. The parameter may be used to help identify timers that use the same expiration routine.
initial_time	Specifies the initial number of timer ticks for timer expiration.
reschedule_time	Specifies the number of timer ticks for expiration after the first expiration. If this parameter is zero, the timer only expires once.
enable	Valid options for this parameter are NU_ENABLE_TIMER and NU_DISABLE_TIMER. NU_ENABLE_TIMER activates the timer after it is created. NU_DISABLE_TIMER leaves the timer disabled. Timers created with the NU_DISABLE_TIMER must be enabled by a call to NU_Control_Timer at a later time.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_TIMER	Indicates the timer control block pointer is NULL or is already in use.
NU_INVALID_FUNCTION	Indicates the expiration function pointer is NULL.
NU_INVALID_ENABLE	Indicates the enable parameter is invalid.

Example

```

/* Assume timer control block "Timer" is defined as a
global data structure. This is one of several ways
to allocate a control block. */

NU_TIMER Timer;
.
.
/* Assume status is defined locally. */

STATUS      status;      /* Timer creation status */

/* Create a timer that has an expiration function "timer_expire",
an ID of 0, an initial expiration of 23 timer ticks. After
the initial expiration, the timer expires every 5 timer ticks.
Note that the timer is enabled during creation. */
status = NU_Create_Timer(&Timer,"any name", timer_expire,
                        0, 23, 5, NU_ENABLE_TIMER);

/* At this point status indicates if the service was successful. */

```

See Also

```

NU_Delete_Timer, NU_Established_Timers, NU_Timer_Pointers,
NU_Timer_Information, NU_Reset_Timer

```

NU_Delete_Timer

```
STATUS NU_Delete_Timer(NU_TIMER *timer)
```

This service deletes a previously created application timer. The parameter `timer` identifies the timer to delete. **Note:** the specified timer must be disabled prior to this service request. The application must prevent the use of this timer during and after deletion.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, task
Category	Timer Services

Parameters

Parameter	Meaning
<code>timer</code>	Pointer to the user-supplied timer control block.

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_TIMER</code>	Indicates the timer pointer is invalid.
<code>NU_NOT_DISABLED</code>	Indicates the specified timer is not disabled.



Example

```
NU_TIMER    Timer;
STATUS      status
.
.
.
/* Delete the timer control block "Timer". Assume "Timer"
   has previously been created with the Nucleus PLUS
   NU_Create_Timer service call.  */
status =  NU_Delete_Timer(&Timer);

/* At this point, status indicates whether the service
   request was successful.  */
```

See Also

NU_Create_Timer, NU_Established_Timers, NU_Timer_Pointers,
NU_Timer_Information, NU_Reset_Timer



NU_Established_Timers

UNSIGNED NU_Established_Timers(VOID)

This service returns the number of established timers. All created timers are considered established. Deleted timers are no longer considered established.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, task
Category	Timer Services

Parameters

None

Return Value

This service returns the number of established timers.

Example

```
UNSIGNED total_timers;

/* Obtain the total number of timers. */
total_timers = NU_Established_Timers( );
```

See Also

NU_Create_Timer, NU_Delete_Timer, NU_Timer_Pointers,
NU_Timer_Information, NU_Reset_Timer



NU_Reset_Timer

```
STATUS NU_Reset_Timer(NU_TIMER *timer,
                     VOID (*expiration_routine)(UNSIGNED),
                     UNSIGNED initial_time,
                     UNSIGNED reschedule_time,
                     OPTION enable)
```

This service resets the specified timer with new operating parameters. **Note:** the timer must be disabled before this service is called.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, task
Category	Timer Services

Parameters

Parameter	Meaning
timer	Pointer to the timer.
expiration_routine	Specifies the application routine to execute when the timer expires.
initial_time	Specifies the initial number of timer ticks for timer expiration.
reschedule_time	Specifies the number of timer ticks for expiration after the first expiration. If this parameter is zero, the timer only expires once.
enable	Valid options for this parameter are NU_ENABLE_TIMER and NU_DISABLE_TIMER. NU_ENABLE_TIMER activates the timer immediately after it is reset. NU_DISABLE_TIMER leaves the timer disabled. Timers reset with NU_DISABLE_TIMER must be enabled by a call to NU_Control_Timer at a later time.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_TIMER	Indicates the timer control block pointer is invalid.
NU_INVALID_FUNCTION	Indicates the expiration function pointer is NULL.
NU_INVALID_ENABLE	Indicates the enable parameter is invalid.
NU_NOT_DISABLED	Indicates the timer is currently enabled. It must be disabled before it can be reset.

Example

```

NU_TIMER    Timer;
STATUS      status
.
.
.
/* Reset the timer control block "Timer" to expire initially
   after 3 timer ticks and then expire every 30 timer ticks.
   Also, the new expiration routine is "new_expire".
   Automatically enable the timer after it is reset. Assume
   "Timer" has previously been created with the Nucleus PLUS
   NU_Create_Timer service call. */
status = NU_Reset_Timer(      &Timer, new_expire, 3, 30,
NU_ENABLE_TIMER);

/* Contents of status indicates whether or not the
   service was successful. */

```

See Also

```

NU_Create_Timer, NU_Delete_Timer, NU_Control_Timer,
NU_Timer_Information

```



NU_Retrieve_Clock

UNSIGNED NU_Retrieve_Clock(VOID)

This service returns the current value of the continuously incrementing timer tick counter. The counter increments once for every timer interrupt.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, task
Category	Timer Services

Parameters

None

Return Value

This service call returns the current value of the system clock.

Example

```

UNSIGNED  clock_value;

/* Read the current value of the system tick clock. */
clock_value = NU_Retrieve_Clock( );

```

See Also

NU_Set_Clock

NU_Set_Clock

```
VOID NU_Set_Clock(UNSIGNED new_value)
```

This service sets the continuously counting system clock to the value specified by `new_value`.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, task
Category	Timer Services

Parameters

Parameter	Meaning
<code>new_value</code>	The new value for the system clock.

Return Value

None

Example

```
/* Set the system clock to 0. */
NU_Set_Clock(0);
```

See Also

NU_Retrieve_Clock

NU_Timer_Information

```
STATUS NU_Timer_Information(NU_TIMER *timer,
                           CHAR *name, OPTION *enable,
                           UNSIGNED *expirations, UNSIGNED *id,
                           UNSIGNED *initial_time,
                           UNSIGNED *reschedule_time)
```

This service returns various information about the specified application timer.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, task
Category	Timer Services

Parameters

Parameter	Meaning
timer	Pointer to the application timer.
name	Pointer to an 8 character destination area for the timer's name.
enable	Pointer to a variable to hold the timer's current enable state, either NU_ENABLE_TIMER or NU_DISABLE_TIMER.
expirations	Pointer to a variable to hold the number of times the timer has expired.
id	Pointer to a variable to hold the user-supplied id.
initial_time	Pointer to a variable to hold the initial timer expiration value.
reschedule_time	Pointer to a variable to hold the timer's reschedule value.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_TIMER	Indicates the timer pointer is invalid.

Example

```
NU_TIMER    Timer;
CHAR        timer_name[8];
OPTION      enable;
UNSIGNED    expirations;
UNSIGNED    id;
UNSIGNED    initial_time;
UNSIGNED    reschedule_time;
STATUS      status;
.
.
.
/* Obtain information about the timer control block "Timer".
Assume "Timer" has previously been created with the
Nucleus PLUS NU_Create_Timer service call. */
status = NU_Timer_Information(&Timer, timer_name, &enable,
                             &expiration, &id, &initial_time,
                             &reschedule_time);

/* If status is NU_SUCCESS, the other information is accurate. */
```

See Also

NU_Create_Timer, NU_Delete_Timer, NU_Established_Timers,
NU_Timer_Pointers, NU_Reset_Timer



NU_Timer_Pointers

```
UNSIGNED NU_Timer_Pointers(NU_TIMER **pointer_list,
                          UNSIGNED maximum_pointers)
```

This service builds a sequential list of pointers to all established application timers in the system. **Note:** timers that have been deleted are no longer considered established. The parameter `pointer_list` points to the location for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, task
Category	Timer Services

Parameters

Parameter	Meaning
<code>pointer_list</code>	Pointer to an array of <code>NU_TIMER</code> pointers. This array will be filled with pointers of established timers in the system.
<code>maximum_pointers</code>	The maximum number of <code>NU_TIMER</code> pointers to place into the array. Typically this will be the size of the <code>pointer_list</code> array.

Return Value

This service call returns the number of timers that are active in the system.

Example

```
/* Define an array capable of holding 20 timer pointers. */
NU_TIMER    *Pointer_Array[20];
UNSIGNED    number;

/* Obtain a list of currently active timer pointers
   (Maximum of 20). */
number = NU_Timer_Pointers(&Pointer_Array[0], 20);

/* At this point, number contains the actual number
   of pointers in the list. */
```

See Also

```
NU_Create_Timer, NU_Delete_Timer, NU_Established_Timers,
NU_Timer_Information, NU_Reset_Timer
```



Example Source Code

The following example program demonstrates how a Nucleus PLUS timer could be used to execute code on a periodic basis. The following Nucleus PLUS program contains a single timer that expires every 5 timer ticks.

Include all necessary Nucleus PLUS include files.

```
#include "nucleus.h"
```

A single Nucleus PLUS structure is required for this demonstration. The timer control block, `NU_TIMER`, will be associated with a timer expiration routine using the `NU_Create_Timer` service call.

```
NU_TIMER timer_demo;
```

The function `expiration_routine` will serve as the timer expiration routine for the `timer_demo` timer. The only parameter necessary for a timer expiration routine is a single `UNSIGNED` which will contain the timer id for which this timer was associated with in the `NU_Create_Timer` service call. As an example, this id could be used to allow the same expiration routine to be used for multiple timers.

```
void expiration_routine(UNSIGNED id);
```

In this demonstration, the `Application_Initialize` function will be used to create the single Nucleus PLUS timer. After `Application_Initialize` executes, all tasks will be created, and the system will be ready to begin executing in a multi-tasking environment.

```
VOID Application_Initialize(VOID *first_available_memory)
{
```

Create the Nucleus PLUS timer with the `NU_Create_Timer` service call. The timer, `timer_demo` will be named "TIMER", and will be associated with the timer expiration routine, `expiration_routine`. The timer will be given the id of 1, will expire 5 timer ticks after processing begins, and will expire every 5 timer ticks thereafter. The `NU_ENABLE_TIMER` parameter specifies that this timer should be immediately enabled. The parameter `NU_DISABLE_TIMER` could also be used, which would require that the function `NU_Control_Timer` was issued later to begin timer processing. The use of this method would allow for timers to be enabled and disabled based upon the current status of the system. Similarly, `NU_Reset_Timer` could also be used to later modify the functionality of the timer.

```
NU_Create_Timer(&timer_demo, "TIMER", expiration_routine, 1, 5, 5,  
NU_ENABLE_TIMER);  
}
```

The function `expiration_routine` is the function that will be executed whenever the `timer_demo` expires. This function will be associated with `timer_demo` using the `NU_Create_Timer` service call.

```
void expiration_routine(UNSIGNED id)  
{  
}
```



13

Interrupts

Introduction
Function Reference
Managed ISRs
Unmanaged ISRs



Introduction

An interrupt is a mechanism for providing immediate response to an external or internal event. When an interrupt occurs, the processor suspends the current path of execution and transfers control to the appropriate Interrupt Service Routine (ISR). The exact operation of an interrupt is inherently processor-specific.

Nucleus PLUS supports both managed and unmanaged ISRs. A managed ISR is one that does not need to save and restore context, while an unmanaged ISR is fully responsible for saving and restoring any registers used. Managed ISRs may be written in C or assembly language. However, unmanaged ISRs are almost always written in assembly language.

Protection

Interrupts pose interesting problems for all real-time kernels. Nucleus PLUS is no exception. The main problem stems from the fact that ISRs need to have access to Nucleus PLUS services. On the surface this may not seem like a problem; however, it requires protection of data structures manipulated during a service call from simultaneous access by an ISR. The simplest method of protection is to lock out interrupts for the duration of the service.

Responding to interrupts quickly is a cornerstone of real-time systems. Therefore, locking out interrupts to protect internal data structures is not desirable. Nucleus PLUS handles this protection problem by dividing application ISRs into low and high-level components.

Low-Level ISR

The Low-Level Interrupt Service Routine (LISR) executes as a normal ISR, which includes using the current stack. Nucleus PLUS saves context before calling an LISR and restores context after the LISR returns. Therefore, LISRs may be written in C and may call other C routines. However, there are only a few Nucleus PLUS services available to an LISR. If the interrupt processing requires additional Nucleus PLUS services, a High-Level Interrupt Service Routine (HISR) must be activated. Nucleus PLUS supports nesting of multiple LISRs.

High-Level ISR

HISRs are created and deleted dynamically. Each HISR has its own stack space and its own control block. The memory for each is supplied by the application. Of course, the HISR must be created before it is activated by an LISR.

Since an HISR has its own stack and control block, it can be temporarily blocked if it tries to access a Nucleus PLUS data structure that is already being accessed.

HISRs are allowed access to most Nucleus PLUS services, with the exception of self-suspension services. Additionally, since an HISR cannot suspend on a Nucleus PLUS service, the “suspend” parameter must always be set to `NU_NO_SUSPEND`.



There are three priority levels available to HISRs. If a higher priority HISR is activated during processing of a lower priority HISR, the lower priority HISR is preempted in much the same manner as a task gets preempted. HISRs of the same priority are executed in the order in which they were originally activated. All activated HISRs are processed before normal task scheduling is resumed.

An activation counter is maintained for each HISR. This counter is used to insure that each HISR is executed once for each activation. Note that each additional activation of an already active HISR is processed by successive calls to that HISR.

HISR Information

Application tasks may obtain a list of active HISRs. Detailed information about each HISR is also available. This information includes the HISR name, total scheduled count, priority, and stack parameters.

Interrupt Latency

Interrupt latency is a term that describes the amount of time for which interrupts are locked out. Since Nucleus PLUS does not rely on locking out interrupts to protect against simultaneous ISR access, interrupt latency is small and constant. In fact, interrupts are only locked out over several instructions in some Nucleus PLUS ports.

Application Interrupt Lockout

Applications are provided with the ability to disable and enable interrupts. An interrupt locked out by the application remains locked out until the application unlocks it.

Direct Vector Access

Nucleus PLUS provides the ability to directly set up interrupt vectors. ISRs loaded directly into the vector table are required to save and restore registers used. Therefore, ISRs entered directly into the vector table are often written in assembly language. Such ISRs, providing certain conventions are followed, may activate a HISR.



Function Reference

The following function reference contains all functions related to Nucleus PLUS timers.
The following functions are contained in this reference:

- NU_Activate_HISR
- NU_Control_Interrupts
- NU_Create_HISR
- NU_Current_HISR_Pointer
- NU_Delete_HISR
- NU_Established_HISRs
- NU_HISR_Information
- NU_HISR_Pointers
- NU_Local_Control_Interrupts
- NU_Register_HISR
- NU_Setup_Vector



NU_Activate_HISR

```
STATUS NU_Activate_HISR (NU_HISR *histr)
```

This service activates the HISR pointed to by `histr`. If the specified HISR is currently executing, this activation request is not processed until the current execution is complete. A HISR is executed once for each activation request.

Overview

Option	
Tasking Changes	No
Allowed From	LISR
Category	Interrupt Services

Parameters

Parameter	Meaning
<code>histr</code>	Pointer to the user-supplied HISR control block.

Return Value

A return value of `NU_SUCCESS` indicates successful completion of this service.

Example

```
NU_HISR    Operator_Input;
STATUS     status;

/* Activate the previously created operator input HISR
   for which the control block is Operator_Input. */
status = NU_Activate_HISR(&Operator_Input);
```

See Also

`NU_Create_HISR`, `NU_Delete_HISR`, `NU_HISR_Information`

NU_Control_Interrupts

```
INT NU_Control_Interrupts(INT new_level)
```

This service enables or disables interrupts according to the value specified in `new_level`. Interrupts are disabled and enabled in a task-independent manner. Therefore, an interrupt disabled by this service remains disabled until enabled by a subsequent call to this service. Values of `new_level` are processor dependent. However, the values `NU_DISABLE_INTERRUPTS` and `NU_ENABLE_INTERRUPTS` may be used to disable all interrupts and enable all interrupts, respectively.

Overview

Option	
Tasking Changes	No
Allowed From	LISR, HISR, Signal Handler, Task
Category	Interrupt Services

Parameters

Parameter	Meaning
<code>new_level</code>	New interrupt level for the system. The options <code>NU_DISABLE_INTERRUPTS</code> (disable all interrupts) and <code>NU_ENABLE_INTERRUPTS</code> (enable all interrupts) are always available. Other options may be available depending upon architecture. See the target specific notes for more information.

Return Value

This service returns the previous level of enabled interrupts.

Example

```
INT old_level;      /* Old interrupt level. */

/* Lockout all interrupts temporarily. */
old_level = NU_Control_Interrupts(NU_DISABLE_INTERRUPTS);
.
.
.
/* Restore previous interrupt lockout level. */
NU_Control_Interrupts(old_level);
```

See Also

`NU_Setup_Vector`, `NU_Register_LISR`, `NU_Create_HISR`, `NU_Delete_HISR`



NU_Create_HISR

```
STATUS NU_Create_HISR(NU_HISR *hisr, CHAR *name,
                     VOID (*hisr_entry)(VOID),
                     OPTION priority,
                     VOID *stack_pointer,
                     UNSIGNED stack_size)
```

This service creates a High-Level Interrupt Service Routine (HISR). HISRs are allowed to call most Nucleus PLUS services, unlike Low-Level Interrupt Service Routines (LISRs).

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Interrupt Services

Parameters

Parameter	Meaning
hisr	Pointer to the user-supplied HISR control block. Note: all subsequent requests made to this HISR require this pointer.
name	Pointer to an 8 character name for the HISR. The name does not have to be null-terminated.
hisr_entry	Specifies the function entry point of the HISR.
priority	There are three HISR priorities (0-2). Priority 0 is the highest.
stack_pointer	Pointer to the HISR's stack area. Each HISR has its own stack area. Note that the HISR stack is pre-allocated by the caller.
stack_size	Number of bytes in the HISR stack.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_HISR	Indicates the HISR control block pointer is NULL or is already in use.
NU_INVALID_ENTRY	Indicates the HISR entry pointer is NULL.
NU_INVALID_PRIORITY	Indicates the HISR priority is invalid.
NU_INVALID_MEMORY	Indicates the stack pointer is NULL.
NU_INVALID_SIZE	Indicates the stack size is too small.

Example

```
/* Assume HISR control block "HISR" is defined as a global
   data structure. This is one of several ways to allocate
   a control block. */

NU_HISR      HISR
.
.
/* Assume status is defined locally. */

STATUS      status; /* HISR creation status */

/* Create an HISR. Note that the HISR entry function is
   "HISR_Entry" and the "stack_pointer" points to a previously
   allocated block of memory that contains 400 - bytes. */
status = NU_Create_HISR(&HISR, "any name", HISR_Entry,
                        2, stack_pointer, 400);

/* status indicates if the service was successful. */
```

See Also

NU_Delete_HISR, NU_Established_HISRs, NU_HISR_Pointers,
NU_HISR_Information



NU_Current_HISR_Pointer

NU_HISR *NU_Current_HISR_Pointer(VOID)

This service returns the currently executing HISR's pointer. If the caller is not an HISR, the value returned is NU_NULL.

Overview

Option	
Tasking Changes	No
Allowed From	HISR, LISR
Category	Interrupt Services

Parameters

None

Return Value

This service call returns a pointer the currently executing HISR's control block.

Example

```
NU_HISR *HISR_ptr;

/* Get the currently running HISR pointer. */
HISR_ptr = NU_Current_HISR_Pointer( );
```

See Also

NU_Established_HISRs, NU_HISR_Pointers, NU_HISR_Information

NU_Delete_HISR

```
STATUS NU_Delete_HISR(NU_HISR *hisr)
```

This service deletes a previously created HISR. The parameter `hisr` identifies the HISR to delete. The application must prevent the use of this HISR during and after deletion.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, Signal Handler, Task
Category	Interrupt Services

Parameters

Parameter	Meaning
<code>hisr</code>	Pointer to the user-supplied HISR control block.

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_HISR</code>	Indicates the HISR pointer is invalid.

Example

```
NU_HISR  Hisr;
STATUS   status
.
.
.
/* Delete the HISR control block "Hisr". Assume "Hisr"
has previously been created with the Nucleus PLUS
NU_Create_HISR service call. */

status =  NU_Delete_HISR(&Hisr);

/* At this point, status indicates whether the service
request was successful.  */
```

See Also

`NU_Create_HISR`, `NU_Established_HISRs`, `NU_HISR_Pointers`,
`NU_HISR_Information`



NU_Established_HISRs

UNSIGNED NU_Established_HISRs(VOID)

This service returns the number of established HISRs. All created HISRs are considered established. Deleted HISRs are no longer considered established.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, Signal Handler, Task
Category	Interrupt Services

Parameters

None

Return Value

This service call returns the number of established HISRs in the system.

Example

```

UNSIGNED total_hisrs;

/* Obtain the total number of HISRs. */
total_hisrs = NU_Established_HISRs( );

```

See Also

NU_Create_HISR, NU_Delete_HISR, NU_HISR_Pointers,
NU_HISR_Information

NU_HISR_Information

```
STATUS NU_HISR_Information(NU_HISR *hisr, char *name,
                          UNSIGNED *scheduled_count,
                          DATA_ELEMENT *priority,
                          VOID **stack_base,
                          UNSIGNED *stack_size,
                          UNSIGNED *minimum_stack)
```

This service returns various information about the specified HISR.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, Signal Handler, Task
Category	Interrupt Services

Parameters

Parameter	Meaning
hisr	Pointer to the HISR.
name	Pointer to an 8 character destination area for the HISR's name.
scheduled_count	Pointer to a variable for holding the total number of times this HISR has been scheduled.
priority	Pointer to a variable for holding the HISR's priority.
stack_base	Pointer to a pointer for holding the original stack pointer. This is the same pointer supplied during creation of the HISR.
stack_size	Pointer to a variable for holding the total size of the HISR's stack.
minimum_stack	Pointer to a variable for holding the minimum amount of available stack space detected during HISR execution.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_HISR	Indicates the HISR pointer is invalid.

Example

```

NU_HISR          Hisr;
CHAR             hisr_name[8];
UNSIGNED         activations;
DATA_ELEMENT     priority;
VOID             *stack_base;
UNSIGNED         stack_size;
UNSIGNED         minimum_stack;
STATUS           status
.
.
.
/* Obtain information about the HISR control block
   "Hisr". Assume "Hisr" has previously been created
   with the Nucleus PLUS NU_Create_HISR service call. */
status = NU_HISR_Information(&Hisr, hisr_name, &activations,
                           &priority, &stack_base, &stack_size,
                           &minimum_stack);

/* If status is NU_SUCCESS, the other information is accurate. */

```

See Also

NU_Create_HISR, NU_Delete_HISR, NU_Established_HISRs,
NU_HISR_Pointers

NU_HISR_Pointers

```
UNSIGNED NU_HISR_Pointers(NU_HISR **pointer_list,
                          UNSIGNED maximum_pointers)
```

This service builds a sequential list of pointers to all established HISRs in the system. **Note:** HISRs that have been deleted are no longer considered established. The parameter `pointer_list` points to the location used for building the list of pointers, while `maximum_pointers` indicates the maximum size of the list. This service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, Signal Handler, Task
Category	Interrupt Services

Parameters

Parameter	Meaning
<code>pointer_list</code>	Pointer to an array of NU_HISR pointers. This array will be filled with pointers of established HISRs in the system.
<code>maximum_pointers</code>	The maximum number of NU_HISR pointers to place into the array. Typically this will be the size of the <code>pointer_list</code> array.

Return Value

This service call returns the number of HISRS that are active in the system.

Example

```
/* Define an array capable of holding 20 HISR pointers. */
NU_HISR *Pointer_Array[20];
UNSIGNED number;

/* Obtain a list of currently active HISR pointers (Maximum of 20).
*/
number = NU_HISR_Pointers(&Pointer_Array[0],20);

/* At this point, number contains the actual number of
pointers in the list. */
```

See Also

NU_Create_HISR, NU_Delete_HISR, NU_Established_HISRs,
NU_HISR_Information

NU_Local_Control_Interrupts

```
INT NU_Local_Control_Interrupts(INT new_level)
```

This service enables or disables interrupts according to the value specified in `new_level`. Interrupts are disabled and enabled in a subroutine-dependent manner. This service changes the Status Register to the value specified. The Status Register will be set back to value set by the last call to `NU_Control_Interrupts` on the next context switch. Values of `new_level` are processor dependent. However, the values `NU_DISABLE_INTERRUPTS` and `NU_ENABLE_INTERRUPTS` may be used to disable all interrupts and enable all interrupts, respectively.

Overview

Option	
Tasking Changes	No
Allowed From	LISR, HISR, Signal Handler, Task
Category	Interrupt Services

Parameters

Parameter	Meaning
<code>new_level</code>	New interrupt level for the current subroutine. The options <code>NU_DISABLE_INTERRUPTS</code> (disable all interrupts) and <code>NU_ENABLE_INTERRUPTS</code> (enable all interrupts) are always available. Other options may be available depending upon architecture. See the target specific notes for more information.

Return Value

This service returns the previous level of enabled interrupts.

Example

```
INT old_level; /* Old interrupt level. */

/* Lockout all interrupts temporarily. */
old_level=NU_Local_Control_Interrupts(NU_DISABLE_INTERRUPTS);
.
.
.
return; /* Or interrupt return. */
```

See Also

`NU_Setup_Vector`, `NU_Register_LISR`, `NU_Create_HISR`, `NU_Delete_HISR`



NU_Register_LISR

```
STATUS NU_Register_LISR(INT vector, VOID(*lISR_entry)(INT),
                      VOID (**old_lISR)(INT))
```

This service associates the LISR function pointed to by `lISR_entry` with the interrupt vector specified by `vector`. System context is automatically saved before calling the specified LISR and is restored after the LISR returns. Therefore, LISR functions may be written in C. However, LISRs are permitted access to only a few of Nucleus PLUS services. If interaction with other Nucleus PLUS services is required, a High-Level Interrupt Service Routine (HISR) must be activated by the LISR.

If the `lISR_entry` parameter is `NU_NULL`, the registration of the specified vector is cleared.

Caution: If an LISR is written in assembly language, it must follow the C compiler's conventions regarding register usage and the return mechanism. See your compiler documentation for specific requirements of C-assembly language interaction.

Overview

Option	
Tasking Changes	No
Allowed From	LISR, HISR, Signal Handler, Task
Category	Interrupt Services

Parameters

Parameter	Meaning
<code>vector</code>	The interrupt vector at which to register the interrupt.
<code>lISR_entry</code>	The subroutine to register at the vector.
<code>old_lISR</code>	The subroutine previously registered at the specified vector.



Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_VECTOR	Indicates the specified vector is invalid.
NU_NOT_REGISTERED	Indicates the vector is not currently registered and de-registration was specified by <code>l_isr_entry</code> .
NU_NO_MORE_LISRS	Indicates the maximum number of registered LISRs has been exceeded. The maximum number can be changed in <code>NUCLEUS.H</code> . Note: Given this return value, the Nucleus PLUS library will need to be rebuilt.

Example

```

STATUS      status;
VOID        (*old_lisr)(INT);

/* Associate vector 10 with the LISR function "LISR_example".

void LISR_example(INT vector_number)

/* vector_number contains the actual interrupt
vector number. */

/* Nucleus PLUS service calls, with the exception of
NU_Activate_HISR and several others, are not allowed in this
function. */

status = NU_Register_LISR(10, LISR_example, &old_lisr);

/* If status is NU_SUCCESS, LISR_example is executed when
interrupt vector 10 occurs. Note: "old_lisr" contains
the previously registered LISR. */

```

See Also

`NU_Control_Interrupts`, `NU_Create_HISR`, `NU_Delete_HISR`,
`NU_Activate_HISR`



NU_Setup_Vector

```
VOID *NU_Setup_Vector(INT vector, VOID *new)
```

This service replaces the interrupt vector specified by `vector` with the custom Interrupt Service Routine (ISR) supplied by the caller (parameter `new`). The previous interrupt vector contents are returned by the service.

Caution: ISRs supplied to this routine are typically written in assembly language and are responsible for saving and restoring any registers used. In some ports of Nucleus PLUS there are some additional constraints imposed on such ISRs. Please see the processor-specific Portation Notes for additional target-specific information.

Overview

Option	
Tasking Changes	No
Allowed From	LISR, HISR, Signal Handler, Task
Category	Interrupt Services

Parameters

Parameter	Meaning
<code>vector</code>	The interrupt vector at which to register the interrupt.
<code>new</code>	The subroutine to register at the vector.

Return Value

A pointer to the subroutine previously registered at the interrupt vector.

Example

```
VOID *old_vector;

/* Place an assembly language ISR named "asm_ISR" into vector 5. */
old_vector = NU_Setup_Vector(5, asm_ISR);
```

See Also

NU_Control_Interrupts, NU_Register_LISR



Managed ISRs

Managed ISRs are referred to in this document as Low-Level Interrupt Service Routines (LISR). LISRs execute in the same fashion as a traditional ISR, except all context saving and restoring is taken care of by Nucleus PLUS.

The following is an example segment of code that defines a LISR function and registers it with vector 10:

```
VOID      (*old_lisr)(INT);
VOID      Example_LISR(INT vector);
INT       Interrupt_Count = 0;
.
.
/* Register the LISR with vector 10. The previously registered
   LISR is returned in old_lisr. */
NU_Register_LISR(10, Example_LISR, &old_lisr);
.
.
.

/* Actual definition of the LISR associated with
   vector 10. */
VOID Example_LISR(INT vector)
{
    /* Increment the global interrupt counter. */
    Interrupt_Count++;
}
```

When interrupt 10 occurs, `Example_LISR` is called with the vector parameter set to 10. Interrupt processing consists of incrementing a global variable, which is completed when `Example_LISR` returns. It is important to note that LISRs have extremely limited access to Nucleus PLUS services. For example, if a task must be resumed as a result of interrupt 10, a High-Level Interrupt Service Routine (HISR) must be activated from within the LISR.

The following example resumes the task pointed to by `Task_0_Ptr` when interrupt 10 occurs:

```
extern NU_TASK    *Task_0_Ptr;
NU_HISR          HISR_Control;
CHAR             HISR_Stack[500];
VOID              (*old_lisr)(INT);
VOID              Example_LISR(INT vector);
VOID              Example_HISR(VOID);
.
.
.

/* Create a HISR. This HISR is activated by the LISR
   associated with vector 10. */
NU_Create_HISR(&HISR_Control, "EXMPHISR",
Example_HISR, 2, HISR_Stack, 500);
```



```
/* Register the LISR with vector 10. The previously
   registered LISR is returned in old_lisr. */

NU_Register_LISR(10, Example_LISR, &old_lisr);
.
.
.
/* Actual definition of the LISR associated with
   vector 10. */
VOID      Example_LISR(INT vector)
{

    /* Activate Example_HISR to resume the task pointed to by
       "Task_0_Ptr." Not allowed to call most Nucleus PLUS
       services from LISR. */

    NU_Activate_HISR(&HISR_Control);

}

/* Actual definition of the HISR associated with the
   Example_LISR function. */
VOID      Example_HISR(void)
{
    /* Resume the task pointed to by "Task_0_Ptr" */

    NU_Resume_Task(Task_0_Ptr);

}
```



Unmanaged ISRs

Nucleus PLUS supports unmanaged ISRs through direct access to the interrupt vector table (in most processor architectures). The `NU_Setup_Vector` service may be used to associate a specific interrupt vector with the unmanaged ISR. Alternatively, the unmanaged ISR's address may be placed directly in the Nucleus PLUS vector table, which is usually defined in the `INT.?` file.

Unmanaged ISRs are typically implemented for high-frequency interrupts. The amount of overhead associated with context saving and restoring is proportional to frequency of the interrupts. When the time between interrupts gets anywhere near the time required to save and restore context, an unmanaged ISR is necessary. For example, if an interrupt occurs every $30\mu\text{s}$ and managed interrupts require $15\mu\text{s}$ of overhead, half of the processing power is lost in the management of the interrupt.

Suppose a *mythical* processor has 32 registers, named `r0..r31`. Now suppose that every $30\mu\text{s}$ an interrupt occurs. Furthermore, the only requirement of the ISR is to place a 1 in some memory-mapped location. The following is an example of a minimal ISR (in mythical assembly language) to satisfy the requirement:

Minimal_ISR:

```
push r0                ; Save r0
mov 1, r0              ; Place a 1 into r0
mov r0, mem_map_loc    ; Set memory mapped location
pop r0                ; Recover r0
iret                  ; Return from interrupt
```

If a fully managed interrupt on this mythical processor requires $15\mu\text{s}$ to save and restore all 32 registers, and this minimal ISR only takes $1\mu\text{s}$, then a $30\mu\text{s}$ interrupt might be feasible.

Unfortunately, not all high-frequency interrupt handlers are so easy. In many situations, such interrupts correspond to the availability of data to process. The most common technique to handle this situation involves buffering the interrupt information. The minimal ISR manipulates data in a global memory location for processing by an application task that either runs continuously or in some periodic fashion. An alternative method would be the creation of a minimal ISR that manages buffered data, and occasionally invokes Nucleus PLUS.

The following is the same minimal ISR, with occasional interaction with Nucleus PLUS. (Assume previous LISR/HISR example definitions.)

Minimal_ISR:

```

push r0                ; Save r0
mov 1, r0              ; Place a 1 into r0
mov r0, mem_map_loc    ; Set memory mapped location
                        ; Buffer processing in this area
;
; Check to see if a buffer overflow condition
; is present. If so, invoke Nucleus PLUS to
; wake up task 0.
mov buffer_full, r0     ; Put buffer full code in r0
cmp r0, 1               ; If buffer is not full, just
jne _Fast_Interrupt     ; process fast interrupt
;
; Call Nucleus PLUS context save routine
pop r0                  ; Recover r0
call _TCT_Interrupt_Context_Save
;
mov 10, r0              ; Put vector number into r0
push r0                ; Put it on the stack
call _Example_LISR      ; Call Example_LISR to activate
                        ; HISR that actually resumes
                        ; task 0
pop r0                  ; Clean up the stack
;
; Restore context, note that control does not return
jmp _TCT_Interrupt_Context_Restore
_Fast_Interrupt:
pop r0                  ; Recover r0
iret                    ; Return from interrupt

```

Of course, the previous examples are in an assembly language for a mythical processor. Detailed examples of such interrupt handlers are located in the Portation Notes for the given target processor.



14

System Diagnostics

Introduction

Function Reference

Example Source Code



Introduction

Nucleus PLUS provides application tasks with several facilities that improve diagnosis of system problems.

Error Management

If a fatal system error occurs, processing is transferred to a common error handling routine. By default, this routine prepares an ASCII error message and halts the system. However, additional error processing may be added by the application developer.

System History

Nucleus PLUS provides a circular log of various system activities. Application tasks and HISRs can make entries to this log. Nucleus PLUS services have a conditional compilation option that enables entries into the history log each time a service request is made. Each entry in the history log contains information about the service and the caller.

Version Information

`RLD_Release_String` is a global C string that contains the current version and release of the Nucleus PLUS software. Examination of this string in the target system provides quick identification of the underlying Nucleus PLUS system.

License Information

`LID_License_String` is a global C string that contains customer license information, including the customer's serial number.

Building the PLUS Library

In order for history saving to be enabled, the Nucleus PLUS library must be rebuilt to support history saving. In order to save code space, this feature defaults to off for all Nucleus PLUS libraries. To enable history saving, the library must be built with `NU_ENABLE_HISTORY` defined.



Function Reference

The following function reference contains all functions related to Nucleus PLUS system diagnostics. The following functions are contained in this reference:

```
NU_Disable_History_Saving  
NU_Enable_History_Saving  
NU_Licence_Information  
NU_Make_History_Entry  
NU_Release_Information  
NU_Retrieve_History_Entry
```



NU_Disable_History_Saving

VOID NU_Disable_History_Saving(VOID)

This service disables internal history saving. Often this service is used to disable history saving in preparation for examination of the history log.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Development Services

Parameters

None

Return Value

None

Example

```
/* Disable history saving. */  
NU_Disable_History_Saving( );
```

See Also

NU_Enable_History_Saving, NU_Retrieve_History_Entry



NU_Enable_History_Saving

VOID NU_Enable_History_Saving(VOID)

This service enables internal history saving.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Development Services

Parameters

None

Return Value

None

Example

```
/* Enable internal history. */
NU_Enable_History_Saving( );
```

See Also

NU_Disable_History_Saving, NU_Retrieve_History_Entry,
NU_Make_History_Entry



NU_License_Information

CHAR *NU_License_Information(VOID)

This service returns a pointer to a string that contains the customer's serial number and a small product description. The string is in ASCII format and is NULL terminated.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Development Services

Parameters

None

Return Value

This service call returns a pointer to a string containing a serial number and product description.

Example

```
CHAR *license_string;

/* Obtain a pointer to the customer's license string. */
license_string = NU_License_Information( );
```

See Also

NU_Release_Information



NU_Make_History_Entry

```
VOID NU_Make_History_Entry(UNSIGNED param1,
                          UNSIGNED param2,
                          UNSIGNED param3)
```

This service makes an entry in the system history log if the history log capability is enabled. Otherwise, this service does nothing.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Development Services

Parameters

Parameter	Meaning
param1	First variable to log to the history entry.
param2	Second variable to log to the history entry.
param3	Third variable to log to the history entry.

Return Value

None

Example

```
/* Make an entry in the history log that has the values
   1, 2, and 3 for the parameters. */
NU_Make_History_Entry(1,2,3);
```

See Also

NU_Enable_History_Saving, NU_Disable_History_Saving,
NU_Retrieve_History_Entry

NU_Release_Information

CHAR *NU_Release_Information(VOID)

This service returns a pointer to the Nucleus PLUS release information string. The string is in ASCII format and is NULL terminated.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Development Services

Parameters

None

Return Value

This service call returns a pointer to a string containing release information.

Example

```
CHAR *release_pointer;  
  
/* Point at the Nucleus PLUS release information string. */  
release_pointer = NU_Release_Information( );
```

See Also

NU_License_Information



NU_Retrieve_History_Entry

```
STATUS NU_Retrieve_History_Entry(DATA_ELEMENT *id,
                                UNSIGNED *param1,
                                UNSIGNED *param2,
                                UNSIGNED *param3,
                                UNSIGNED *time,
                                NU_TASK **task,
                                NU_HISR **hisr)
```

This service returns the oldest entry in the system history log. **Note:** It is usually a good idea to disable history saving prior to using this service. History saving must be enabled in order to record history entries. By default, the system history log is disabled at start up.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	Development Services

Parameters

Parameter	Meaning
id	Pointer to a variable for holding the ID of the entry. Note: Nucleus PLUS service IDs are the service name in CAPS with an _ID appended to the end. Entries made by the user have an ID of NU_USER_ID.
param1, 2, 3	Pointers to variables for holding the first, second, and third history parameter entries.
time	Pointer to a variable for holding the value of the system clock that corresponds to this entry.
task	Pointer to a task pointer for holding the pointer of the task that made the entry.
hisr	Pointer to a HISR pointer for holding the pointer of the HISR that made the entry.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_END_OF_LOG	Indicates that there are no more entries in the log.



Example

```
DATA_ELEMENT      id;
UNSIGNED           param1;
UNSIGNED           param2;
UNSIGNED           param3;
UNSIGNED           time;
NU_TASK            *task;
NU_HISR            *hisr;

/* Assume the system history log is already disabled.
   Pick up the next most recent entry. */
status = NU_Retrieve_History_Entry(&id, &param1, &param2,
                                   &param3, &time, &task,
                                   &hisr);

/* If status is NU_SUCCESS, the supplied variables have valid
   information. Note: either task or hisr must be NULL. */
```

See Also

NU_Enable_History_Saving, NU_Disable_History_Saving



Example Source Code

The following example will demonstrate how the Nucleus PLUS history functions could be used to store a log of system errors.

Include all necessary Nucleus PLUS include files

```
#include "nucleus.h"
```

As previously mentioned, this example will demonstrate how history entries could be used to indicate whenever a system error occurred. The following defines will be used to indicate what error has occurred, and will be used by the `NU_Make_History_Entry` service call.

```
#define ERR_CREATE_MEMORY      101
#define ERR_ALLOCATE_MEMORY    102
#define ERR_CREATE_TASK        103
```

Two Nucleus PLUS structures will be used in this example. The task control block, `dm_memory`, will be used for the dynamic memory pool out of which all memory will be allocated. The `NU_TASK` structure, `task_send` will be used in `Application_Initialize` by the `NU_Create_Task` service call.

```
NU_MEMORY_POOL dm_memory;
NU_TASK task_send;
```

A single void pointer will be used in this example. The void pointer will hold a pointer to the task stack for `task_send`. Although not demonstrated in this program, this pointer could be used at a later time in the program to deallocate the stack associated with this task.

```
VOID *stack_send;
```

Two functions will be used in this demonstration program. The function `error` will be used to make a history entry with the `NU_Make_History_Entry` service call. Similarly, the `process_history` function will be used to retrieve the history entries with `NU_Retrieve_History_Entry`.

```
VOID error(UNSIGNED err_code);
VOID process_history();
```

Declare the task entry point function for the `task_send` task. This function will later be passed as a parameter to the `NU_Create_Task` call which will associate it with the `task_send` task control block. For this example, the function `send_data` will serve as a function stub for the `NU_Create_Task` function call.

```
VOID send_data(UNSIGNED argc, VOID *argv);
```

In this demonstration, the `Application_Initialize` function will be used to make several Nucleus PLUS service calls, and check the return values for errors. If an error occurs, a call to the `error` function will be issued, which will in turn call `NU_Make_History_Entry`.



```
VOID Application_Initialize(VOID *first_available_memory)
{
```

We must tell Nucleus PLUS that we will be making history entries. We do this by calling `NU_Enable_History_Saving`. This service call allows an application to be developed that makes numerous history entries. Then, by removing this function call, history processing will not occur, which can save valuable processing time.

Enable application level history saving with a call to `NU_Enable_History_Saving`.

```
NU_Enable_History_Saving();
```

For this example, we will make a call to `NU_Create_Memory_Pool`, `NU_Allocate_Memory`, and `NU_Create_Task`. For each of these calls, check the return status. If an error occurred, make a call to the error function. The error function will then make a call to `NU_Make_History_Entry` to log the error.

```
if (NU_Create_Memory_Pool(&dm_memory, "system",
                        first_available_memory, 10240, 128,
                        NU_FIFO) != NU_SUCCESS)
{
    error(ERR_CREATE_MEMORY);
}

if (NU_Allocate_Memory(&dm_memory, &stack_send, 1024, NU_NO_SUSPEND)
    != NU_SUCCESS)
{
    error(ERR_ALLOCATE_MEMORY);
}

if (NU_Create_Task(&task_send, "SEND", send_data, 0, NU_NULL,
                  stack_send, 1024, 3, 0, NU_PREEMPT, NU_START) != NU_SUCCESS)
{
    error(ERR_CREATE_TASK);
}
```

For this example, at the end of `Application_Initialize`, call the `process_history` function, which will retrieve all current history log entries.

```
process_history();
}
```

The `send_data` function is the task entry point for the `task_send` task. In this example, the task is only created to demonstrate history saving, so there is no processing code contained in the task entry point.

```
VOID send_data(UNSIGNED argc, VOID *argv)
{
}
```

Each application level history entry consists of three separate `UNSIGNED` numbers. For this example, we will use the first to record the error, but set the last two to 0 indicating that they are not being used.



```

VOID error(UNSIGNED err_code)
{
    NU_Make_History_Entry(err_code, 0, 0);
}

```

The function `print_history` will be used to loop through all history entries, removing each of them from the history log.

```

VOID print_history()
{

```

The following data elements will be used by the `NU_Retrieve_History_Entry` service call, and will hold the individual elements for each history entry.

```

DATA_ELEMENT id;
UNSIGNED param1;
UNSIGNED param2;
UNSIGNED param3;
UNSIGNED time;
NU_TASK *task;
NU_HISR *hisr;

```

```

CHAR *license_info;
CHAR *release_info;

```

After the following two service calls, `license_info` and `release_info` will contain a string holding the license information, and release information respectively.

```

license_info = NU_License_Information();
release_info = NU_Release_Information();

```

It is always good practice to disable history saving before retrieving history entries. To do this, call the `NU_Disable_History_Saving` service call.

```

NU_Disable_History_Saving();

```

For each entry in the history log, call `NU_Retrieve_History_Entry` to remove the history entry from the log. The `NU_Retrieve_History_Entry` service call returns `NU_SUCCESS` if a history entry was successfully received, so we will use this to continually loop until there are no more history entries. While not demonstrated here, each history entry could then be sent to a serial port, saved to external storage, or any other means to store the history log.

```

while (NU_Retrieve_History_Entry(&id, &param1, &param2, &param3,
&time, &task, &hisr)
== NU_SUCCESS)
{
}

```

To turn history saving back on, call the `NU_Enable_History_Saving` service call.

```

NU_Enable_History_Saving();
}

```



15

I/O Drivers

Introduction

Function Reference

Implementing an I/O Driver



Introduction

Most real-time applications require input and output with various peripherals. The management of such input and output is usually accomplished with an I/O device driver.

Common Interface

Nucleus PLUS provides a standard I/O driver interface for initialization, assign, release, input, output, status, and terminate requests. This interface is implemented with a common control structure. Each driver has a single point of entry. The control structure identifies the service requested and all necessary parameters. If a specific driver requires additional parameters, the control structure provides a mechanism to link a supplemental control structure to it. Having a standard interface enables applications to deal with a variety of peripherals in a similar, if not identical, manner.

Driver Contents

An I/O driver usually handles processing of initialize, assign, release, input, output, status, and terminate requests. If the I/O driver is interrupt driven, interrupt handling routines are also necessary.

Nucleus PLUS facilities may be used from within the I/O driver. Queues, pipes, and semaphores are commonly utilized by I/O drivers.

Protection

In addition to the availability of most Nucleus PLUS services, I/O drivers are also supplied with a service to protect internal data structures against simultaneous High-Level ISR access. Protection from simultaneous access by Low-Level ISRs is accomplished by disabling the appropriate interrupt.

Suspension

I/O drivers may be called from various threads in the system. If an I/O driver is called from a task thread, suspension facilities associated with other Nucleus PLUS facilities are available. Additionally, a service is provided to suspend and clear the HISR protection simultaneously.

Dynamic Creation

Nucleus PLUS I/O drivers are created and deleted dynamically. There is no preset limit on the number of I/O drivers an application may have. Each I/O driver requires a control block. The control block memory is supplied by the application. Create and delete driver routines do not actually invoke the driver. Separate calls must be made to initialize and terminate the driver.



Driver Information

Application tasks may obtain a list of active I/O drivers. Detailed information is driver-specific.

Function Reference

The following function reference contains all functions related to Nucleus PLUS I/O Drivers. The following functions are contained in this reference:

- NU_Create_Driver
- NU_Delete_Driver
- NU_Driver_Pointers
- NU_Established_Drivers
- NU_Protect
- NU_Request_Driver
- NU_Resume_Driver
- NU_Suspend_Driver
- NU_Unprotect



NU_Create_Driver

```
STATUS NU_Create_Driver(NU_DRIVER *driver, CHAR *name,
                       VOID (*driver_entry)
                       (NU_DRIVER*, NU_DRIVER_REQUEST*))
```

This service creates an Input/Output Driver. **Note:** this service does not invoke the driver.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	I/O Driver Services

Parameters

Parameter	Meaning
driver	Pointer to the user-supplied driver control block. Note: all subsequent requests made to the driver require this pointer.
name	Pointer to an 8 character name for the driver. The name does not have to be null-terminated.
driver_entry	Specifies the function entry point to the driver. Note: the function must conform to the described interface.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_DRIVER	Indicates the driver pointer is NULL or is already in use.
NU_INVALID_POINTER	Indicates the entry pointer is NULL.

Example

```

/* Assume driver's control block "Driver" is defined
   as a global data structure. This is one of
   several ways to allocate a control block. */

NU_DRIVER  Driver;
.
.
.
/* Assume status is defined locally.  */

STATUS      status; /* Driver creation status */

/* Create a driver where the function "Driver_Entry" is
   the entry point. Note that NU_Request_Driver must be
   called after this to actually initialize the I/O
   driver.  */

status =  NU_Create_Driver(&Driver, "any name", Driver_Entry);

/* At this point, status indicates if the service was successful.
   */

```

See Also

NU_Delete_Driver, NU_Established_Drivers, NU_Driver_Pointers

NU_Delete_Driver

```
STATUS NU_Delete_Driver(NU_DRIVER *driver)
```

This service deletes a previously created I/O driver. The parameter `driver` identifies the I/O driver to delete. All usage of the specified driver must be complete prior to calling this service. This is typically accomplished with a terminate request.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	I/O Driver Services

Parameters

Parameter	Meaning
<code>driver</code>	Pointer to the user-supplied driver control block.

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_DRIVER</code>	Indicates the driver pointer is invalid.

Example

```
NU_DRIVER   Driver;
STATUS      status
.
.
.
/* Delete the driver control block "Driver". Assume
   "Driver" has previously been created with the
   Nucleus PLUS NU_Create_Driver service call. */

status = NU_Delete_Driver(&Driver);

/* At this point, status indicates whether the
   service request was successful. */
```

See Also

`NU_Create_Driver`, `NU_Established_Drivers`, `NU_Driver_Pointers`

NU_Driver_Pointers

```
UNSIGNED NU_Driver_Pointers(NU_DRIVER **pointer_list,
                           UNSIGNED maximum_pointers)
```

This service builds a sequential list of pointers to all established I/O drivers in the system. **Note:** I/O drivers that have been deleted are no longer considered established. The parameter `pointer_list` points to the location to build the list of pointers, while `maximum_pointers` indicates the maximum size of the list. The service returns the actual number of pointers in the list. Additionally, the list is ordered from oldest to newest member.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	I/O Driver Services

Parameters

Parameter	Meaning
<code>pointer_list</code>	Pointer to an array of NU_DRIVER pointers. This array will be filled with pointers of established I/O Drivers in the system.
<code>maximum_pointers</code>	The maximum number of NU_DRIVER pointers to place into the array. Typically this will be the size of the <code>pointer_list</code> array.

Return Value

This service call returns the number of HISRS that are active in the system.

Example

```
/* Define an array capable of holding 20 I/O driver pointers. */
NU_DRIVER *Pointer_Array[20];
UNSIGNED number;

/* Obtain a list of currently active I/O drivers (Maximum of 20). */
number = NU_Driver_Pointers(&Pointer_Array[0], 20);

/* At this point, number contains the actual number
   of pointers in the list. */
```

See Also

`NU_Create_Driver`, `NU_Delete_Driver`, `NU_Established_Drivers`



NU_Established_Drivers

UNSIGNED NU_Established_Drivers(VOID)

This service returns the number of established I/O drivers. All created I/O drivers are considered established. Deleted I/O drivers are no longer considered established.

Overview

Option	
Tasking Changes	No
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	I/O Driver Services

Parameters

None

Return Value

This service call returns the number of established I/O Drivers in the system.

Example

```
UNSIGNED total_drivers;

/* Obtain the total number of I/O drivers. */
total_drivers = NU_Established_Drivers( );
```

See Also

NU_Create_Driver, NU_Delete_Driver, NU_Driver_Pointers



NU_Protect

```
VOID NU_Protect(NU_PROTECT *protect_struct)
```

This service initiates primitive protection of a critical data structure. Since I/O Drivers often have to protect against simultaneous access from task and HISR components, this service is typically reserved for protection of data structures within I/O Drivers. Normal task synchronization should be done using the task synchronization services.

Note the following constraints:

The protection structure must be initialized to zero by the application.

After this service is called, the only available Nucleus PLUS services are `NU_Unprotect`, `NU_Suspend_Driver`, and `NU_Resume_Driver`. Nested calls to `NU_Protect` are not allowed.

Overview

Option	
Tasking Changes	Yes
Allowed From	HISR, Task, Signal Handler
Category	I/O Driver Services

Parameters

Parameter	Meaning
<code>protect_struct</code>	Pointer to user supplied protection structure.

Return Value

None

Example

```
NU_PROTECT Protect_Struct;

/* Initiate protection of the critical section associated
   with the protection structure "Protect_Struct."
   Note: Protect_Struct must be cleared prior to first use. */
NU_Protect(&Protect_Struct);
```

See Also

`NU_Unprotect`, `NU_Suspend_Driver`



NU_Request_Driver

```
STATUS NU_Request_Driver(NU_DRIVER *driver,
                        NU_DRIVER_REQUEST *request)
```

This service sends the request structure pointed to by `request` to the I/O Driver specified by `driver`. The definitions of standard I/O Driver requests can be found in Appendix C.

Overview

Option	
Tasking Changes	Yes
Allowed From	Application_Initialize, HISR, Signal Handler, Task
Category	I/O Driver Services

Parameters

Parameter	Meaning
<code>driver</code>	Pointer to the user-supplied driver control block.
<code>request</code>	Pointer to the user-supplied request structure.

Return Value

Status	Meaning
NU_SUCCESS	Indicates successful initiation of the service. The <code>nu_status</code> field of the request structure indicates the actual completion status of the I/O request.
NU_INVALID_DRIVER	Indicates the I/O Driver pointer is invalid.
NU_INVALID_POINTER	Indicates that the I/O request pointer is <code>NULL</code> .



Example

```

NU_DRIVER          Driver;
NU_DRIVER_REQUEST  request;
STATUS             status;
.
.
.
/* Build an initialization request to a simple I/O Driver */
request.nu_function = NU_INITIALIZE;

/* Send the initialization request to "Driver". Assume
   "Driver" has previously been created with the Nucleus PLUS
   NU_Create_Driver service call. */
status = NU_Request_Driver(&Driver, &request);

/* If status indicates success, the driver received the request.
   Additional I/O Driver specific status is available in the
   request structure. */

```

See Also

NU_Established_Drivers, NU_Driver_Pointers



NU_Resume_Driver

```
STATUS NU_Resume_Driver(NU_TASK *task)
```

This service resumes a task previously suspended by an `NU_Suspend_Driver` service. Typically, this service and its suspension counterpart are services used within I/O Drivers. The parameter `task` points to the task to resume.

Overview

Option	
Tasking Changes	Yes
Allowed From	<code>Application_Initialize</code> , HISR, Signal Handler, Task
Category	I/O Driver Services

Parameters

Parameter	Meaning
<code>task</code>	Pointer to the user-supplied task control block.

Return Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_TASK</code>	Indicates the task pointer is invalid.
<code>NU_INVALID_RESUME</code>	Indicates the specified task was not suspended by a <code>NU_Suspend_Driver</code> service.

Example

```
NU_TASK    Task;
STATUS     status;
.
.
.
/* Resume the task control block "Task" that was previously
   suspended by an NU_Suspend_Driver call. Assume "Task" has
   previously been created with the Nucleus PLUS
   NU_Create_Task service call. */
status = NU_Resume_Driver(&Task);
```

See Also

`NU_Suspend_Driver`



NU_Suspend_Driver

```
STATUS NU_Suspend_Driver(VOID (*terminate_routine)(VOID*),
                        VOID *information,
                        UNSIGNED timeout)
```

This service suspends the calling task from within an I/O driver. The termination routine, if specified, allows the driver to clean up any internal structures associated with the calling task during termination or timeout processing. **Note:** any protection established using the `NU_Protect` call is cleared by this service.

Overview

Option	
Tasking Changes	Yes
Allowed From	Task
Category	I/O Driver Services

Return Parameters

Parameter	Meaning
<code>terminate_routine</code>	Pointer to a driver-specific termination/timeout routine (Optional).
<code>information</code>	Pointer to supplemental information required for the termination/timeout routine (Optional).
<code>timeout</code>	Timeout for suspension. A value of <code>NU_SUSPEND</code> indicates an unconditional timeout.

Value

Status	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID_SUSPEND</code>	Indicates that the routine was called from a non-task thread of execution.

Example

```
/* This service is typically used inside of I/O drivers
   to suspend the current task while waiting for I/O.
   Note: any protection established using the NU_Protect
   call is cleared by this service. */
NU_Suspend_Driver(NU_NULL, NU_NULL, 0);
```

See Also

`NU_Resume_Driver`, `NU_Protect`, `NU_Unprotect`



NU_Unprotect

```
VOID NU_Unprotect(VOID)
```

This service lifts the primitive protection of a critical data structure established by a previous call to `NU_Protect`. Since I/O Drivers often have to protect against simultaneous access from task and HISR components, this service is typically reserved for use within I/O drivers. Task synchronization should be done using the task synchronization services.

Note: Care must be taken to avoid calling this routine if protection has already been cleared.

Overview

Option	
Tasking Changes	Yes
Allowed From	HISR, Task
Category	I/O Driver Services

Parameters

None

Return Value

None

Example

```
/* Lift the protection associated with the previous NU_Protect
call. */
NU_Unprotect();
```

See Also

`NU_Protect`, `NU_Suspend_Driver`



Implementing an I/O Driver

Nucleus PLUS provides a basic set of I/O driver facilities. These facilities help foster a consistent driver interface, regardless of the peripheral hardware supported. The basic I/O driver facilities provided with Nucleus PLUS are as follows:

- Create I/O Driver
- Delete I/O Driver
- Request I/O Driver

Before an I/O driver can be used it must be created. This is done with the Nucleus PLUS service call `NU_Create_Driver`. Creation of an I/O driver makes it known to the rest of the system. **Note:** the I/O driver created is not accessed during creation.

An I/O driver may be deleted, if it is no longer needed. The Nucleus PLUS service `NU_Delete_Driver` performs this function. A deleted I/O driver is no longer accessible.

Actual Driver Requests

Applications make requests to drivers using the `NU_Request_Driver` service. The main purpose of this function is to pass the supplied driver request structure to the entry function of the specified I/O driver. The driver request structure contains all the information pertaining to the driver request. This accommodates the different requirements of I/O drivers. For example, an input request made to a disk I/O driver is often different than an input request made to a terminal I/O driver. Typically, a disk driver input request requires a starting location on the disk (sector number) in addition to the number of bytes to read and the buffer pointer. A terminal driver input request does not require any offset information.

Nucleus PLUS has basic support for *initialization*, *assign*, *release*, *input*, *output*, *status*, and *terminate* I/O driver requests. Of course, the parameters of each request may vary, depending on the actual I/O driver.

After the `NU_Request_Driver` service returns, the status of the actual I/O request may be determined by examination of the `nu_status` field in the request structure. If the status field contains `NU_SUCCESS`, the request was completed successfully. If the request was invalid, the contents of the status field is `NU_INVALID_ENTRY`. Finally, if an I/O error is encountered during processing of the request, the `nu_status` field is set to `NU_IO_ERROR`. Additional error information may be added by specific I/O drivers.

Initialization

An initialization request must be made after the I/O driver is created and before any other driver request. This request is used to initialize the managed device and internal driver control structures.



The following is a small code fragment on an initialization request:

```

NU_DRIVER_REQUEST request;
STATUS             status;
.
.
.
/* Build an initialization request for a simple
   I/O driver. */
request.nu_function = NU_INITIALIZE;
request.nu_timeout = NU_NO_SUSPEND;
/* Send the initialization request to the previously
   created I/O driver, pointed to by "driver." */
status = NU_Request_Driver(driver, &request);

/* The variable status indicates whether or not the
   request was passed on to the driver, while the nu_status
   field in the request indicates the completion status of
   the initialization request.*/

```

Assign

An assign request is made in order to prevent simultaneous access to the driver by multiple tasks. For example, if two tasks are sending strings to a terminal handler, one character at a time, the strings are going to be mixed together- resulting in garbage on the screen. If each task obtains exclusive access to the driver before printing the string, this problem is eliminated.

The following is a small code fragment of an assign request.

```

NU_DRIVER_REQUEST      request;
STATUS                 status;
.
.
.
/* Build an assign request for a simple I/O driver.*/
request.nu_function = NU_ASSIGN;
request.nu_timeout = NU_NO_SUSPEND;

/* Send the assign request to the driver pointed to
   by "driver." */
status = NU_Request_Driver(driver, &request);

/* The variable status indicates whether or not the
   request was passed on to the driver, while the
   nu_status field in request indicates the completion
   status of the assign request. */

```



Release

The release request removes a previous assignment. If another task is waiting to assign the driver, the assignment is transferred to the first task waiting. The following is a small code fragment of a release request:

```

NU_DRIVER_REQUEST  request;
STATUS              status;
.
.
.
/* Build a release request for a simple I/O driver.*/
request.nu_function = NU_RELEASE;
request.nu_timeout  = NU_NO_SUSPEND;

/* Send the release request to the driver pointed
   to by "driver." */
status =  NU_Request_Driver(driver, &request);

/* The variable status indicates whether or not the
   request was passed on to the driver, while the nu_
   status field in the request indicates the completion
   status of the release request. */

```

Input

An input request instructs the driver to obtain a certain amount of data from the associated device. The following is a small code fragment of an input request:

```

CHAR                buffer[100];
NU_DRIVER_REQUEST    request;
STATUS               status;
.
.
.
/* Build an input request for a simple I/O driver. */
request.nu_function = NU_INPUT;
request.nu_timeout  = NU_NO_SUSPEND;
request.nu_request_info.nu_input.nu_buffer_ptr = (VOID *) buffer;
request.nu_request_info.nu_input.nu_request_size = 100;

/* Send the input request to the driver pointed to by "driver." */
status =  NU_Request_Driver(driver, &request);
/* If status and request.nu_status are successful, then the buffer
   contains actual data. */

```

Output

An output request instructs the driver to send the specified amount of data to the associated device. The following is a small code fragment of an output request:

```
CHAR                buffer[100];
NU_DRIVER_REQUEST   request;
STATUS              status;
.
.
for /* Build an output request a simple I/O driver.*/
request.nu_function =  NU_OUTPUT;
request.nu_timeout  =  NU_NO_SUSPEND;
request.nu_request_info.nu_output.nu_buffer_ptr =
(VOID *) buffer;
request.nu_request_info.nu_output.nu_request_size = 100;

/* Send the output request to the driver pointed to
   by "driver." */
status =  NU_Request_Driver(driver, &request);

/* If status and request.nu_status are successful, then the buffer
   contents were actually written out. */
```

Status

Status requests are typically I/O driver dependent. The driver's name is always available in the driver control structure, in the field `nu_driver_name`. The following is a small code fragment of a status request:

```
NU_DRIVER_REQUEST   request;
STATUS              status;
.
.
.
/* Build a status request for a simple driver. */
request.nu_function =  NU_STATUS;

/* Send the status request to the driver pointed
   to by "driver." */
status =  NU_Request_Driver(driver, &request);

/* If status is equal to NU_SUCCESS, the driver was
   successfully invoked. The value of request.nu_status,
   along with other possible fields is driver dependent. */
```



Terminate

Terminate requests are typically I/O driver dependent, and are optional. Some drivers may require a terminate request before they can be deleted or re-initialized.

The following is a small code fragment of a terminate request:

```

NU_DRIVER_REQUEST request;
STATUS             status;
.
.
.
/* Build a terminate request for a simple driver. */
request.nu_function = NU_TERMINATE;

/* Send the terminate request to the driver pointed
   to by "driver." */
status = NU_Request_Driver(driver, &request);

/* If status is equal to NU_SUCCESS, the driver was
   successfully terminated. At this point, the driver
   may be deleted or re-initialized. */

```

Driver Implementation

Up to this point, the I/O driver information has been concerned with how to use an I/O driver. This section covers what an I/O driver actually looks like.

I/O drivers are basically a C function with a switch statement. They often include LISR and HISR interrupt handlers and custom functions. All Nucleus PLUS I/O drivers have an entry function similar to the template below:



```

VOID Driver_Entry(NU_DRIVER *driver, NU_DRIVER_REQUEST *request)
{
    /* Process according to the request made. */
    switch(request -> nu_function)
    {
        case    NU_INITIALIZE:
            /* Initialization processing.
            Note: nu_info_ptr field of "driver" is
            available for the driver's use. */
            break;

        case    NU_ASSIGN:
            /* Assign processing. */
            break;

        case    NU_RELEASE:
            /* Release processing. */
            break;

        case    NU_INPUT:
            /* Input processing. */
            break;

        case    NU_OUTPUT:
            /* Output processing. */
            break;

        case    NU_STATUS:
            /* Status processing. */
            break;

        case    NU_TERMINATE:
            /* Terminate processing. */
            break;

        default:
            /* Bad request processing. */
            break;
    }

    /* End of driver request, return to caller. */
}

```

There are several fields available in the driver control structure (NU_DRIVER) to the driver. The following is a list of available structure fields and their associated meaning:

Field	Meaning
nu_info_ptr	Pointer to driver specific information. If used, this field is typically set up during initialization to some type of supplemental control structure specific to the I/O driver.
nu_driver_name	This is the eight-character name associated with the I/O driver.



Example Driver

The code fragment below represents a minimal terminal I/O driver for an MS-DOS system. The driver supports polled, single character input and output requests. **Note:** the driver is accessible only from task threads.

```

/* Entry function of the minimal terminal driver example. */
VOID Terminal_Driver(NU_DRIVER *driver, NU_DRIVER_REQUEST *request)
{
    char *pointer;

    /* Process according to the request made. */
    switch(request -> nu_function)
    {
        case NU_INITIALIZE:
            /* Do nothing for initialization. */
            break;

        case NU_INPUT:
            /* Wait for the user to press a key. */
            while (!kbhit( ))
            {
                /* Sleep a tick to allow other tasks to run. */
                NU_Sleep(1);
            }
            /* Setup input character pointer. */
            pointer = (char *)request ->
            nu_request_info.nu_input.nu_buffer_ptr;
            /* Character present, read it into the
            supplied destination. */
            pointer = (char) getch( );
            /* Indicate successful completion. */
            request -> nu_status = NU_SUCCESS;
            break;

        case NU_OUTPUT:
            /* Setup output character pointer. */
            pointer = (char *) request ->
            nu_request_info.nu_output.nu_buffer_ptr;
            /* Call putch to print supplied character. */
            putch((int) *pointer);
            /* Indicate successful completion. */
            request -> nu_status = NU_SUCCESS;
            break;

        default:
            /* Bad request processing. */
            request -> nu_status = NU_INVALID_ENTRY;
            break;
    }

    /* End of driver request, return to caller. */
}

```





16

Demo Application

Example Overview

Example System



Example Overview

The example system described in this chapter is comprised of an `Application_Initialize` function and six tasks. All of the tasks are created during initialization. In addition to task execution, task communication and synchronization are demonstrated in this example.

In the example system listing, the data structures are defined between lines 5 and 24. Nucleus PLUS control structures are defined between lines 5 and 15.

`Application_Initialize` starts at line 38, and ends at line 90. In this example, all system objects (tasks, queues, semaphores, and event flag groups) are created during initialization. The example system tasks are created between lines 49 and 77. The communication queue is created at line 85. The system semaphore is created at line 82. Finally, the system event flag group is created at line 89. **Note:** a 20,000 byte memory pool, starting at the address specified by the `first_available_memory` parameter is created first, at line 44. This memory pool is used to allocate all of the task stacks and the actual queue area.

Task 0 is the first task to execute when the system starts. This is because task 0 is the highest priority task in the system (priority 1). Task 3 executes after task 0 suspends (priority 5). Task 4 executes after task 3 suspends. It is important to realize why task 3 executes before task 4 even though they both have the same priority. The reason for this is that task 3 was created and started first (see `Application_Initialize`). Tasks of the same priority execute in the order they become ready for execution. After task 4 suspends, task 5 executes (priority 7). After task 5 suspends, task 1 executes (priority 10). Finally, task 2 executes (priority 10) after task 1 suspends on a queue full condition.

Task 0 is defined between lines 100 and 124. Like all of the tasks in this example system, task 0 does some preliminary initialization and then starts execution of an endless loop. Processing inside of task 0's endless loop includes successive calls to `NU_Sleep` and `NU_Set_Events`. Because of the call to `NU_Sleep`, task 0's loop is executed once every 18 timer ticks. **Note:** task 5 is made ready on each call to `NU_Set_Events`. Since task 5 has a lower priority than task 0, it does not execute until task 0 executes the `NU_Sleep` call again.

Task 1 is defined between lines 131 and 162. Task 1 continually sends a single 32-bit message to queue 0. When the capacity of the queue is reached, task 1 suspends, until room is available in queue 0. The suspension of task 1 allows task 2 to resume execution.

Task 2 is defined between lines 169 and 209. Task 2 continually retrieves single 32-bit messages from queue 0. When the queue becomes empty, task 2 suspends. The suspension of task 2 allows task 1 to resume execution.



Tasks 3 and 4 share the same instruction code. However, each task has its own unique stack. Tasks 3 and 4 are defined between lines 218 and 245. Each task competes for a binary semaphore. Once the semaphore is obtained, the task sleeps for 100 ticks before releasing the semaphore again.

This action allows the other task to execute and suspend attempting to obtain the same semaphore. When the semaphore is released, suspension is lifted on the task waiting for the semaphore.

Task 5 is defined between lines 249 and 271. This task is in an endless loop waiting for an event flag to be set. The desired event flag is set by task 0. Therefore, task 5 executes at the same frequency as task 0.

Example System

The following is a source file listing of the example system. **Note:** the line number on the left is not part of the actual file, it is there for reference purposes only.

```

1  /* Include necessary Nucleus PLUS files. */
2  #include "nucleus.h"

3  /* Define Application data structures. */
4  NU_TASK          Task_0;
5  NU_TASK          Task_1;
6  NU_TASK          Task_2;
7  NU_TASK          Task_3;
8  NU_TASK          Task_4;
9  NU_TASK          Task_5;
10 NU_QUEUE         Queue_0;
11 NU_SEMAPHORE      Semaphore_0;
12 NU_EVENT_GROUP    Event_Group_0;
13 NU_MEMORY_POOL    System_Memory;

14 /* Allocate global counters. */
15 UNSIGNED          Task_Time;
16 UNSIGNED          Task_2_messages_received;
17 UNSIGNED          Task_2_invalid_messages;
18 UNSIGNED          Task_1_messages_sent;
19 NU_TASK *Who_has_the_resource;
20 UNSIGNED          Event_Detections;

21 /* Define prototypes for function references. */
22 void task_0(UNSIGNED argc, VOID *argv);
23 void task_1(UNSIGNED argc, VOID *argv);
24 void task_2(UNSIGNED argc, VOID *argv);
25 void task_3_and_4(UNSIGNED argc, VOID *argv);
26 void task_5(UNSIGNED argc, VOID *argv);

27 /* Define the Application_Initialize routine that determines the initial
28 Nucleus PLUS application environment. */

29 void Application_Initialize(void *first_available_memory)
30 {
31 VOID *pointer;

```



```

32 /* Create a system memory pool that will be used to allocate task
33 stacks, queue areas, etc. */
34 NU_Create_Memory_Pool(&System_Memory, "SYSMEM", first_available_memory,
35 20000, 50, NU_FIFO);

36 /* Create each task in the system. */

37 /* Create task 0. */
38 NU_Allocate_Memory(&System_Memory, &pointer, 1000, NU_NO_SUSPEND);
39 NU_Create_Task(&Task_0, "TASK 0", task_0, 0, NU_NULL, pointer, 1000, 1,
40 20, NU_PREEMPT, NU_START);

41 /* Create task 1. */
42 NU_Allocate_Memory(&System_Memory, &pointer, 1000, NU_NO_SUSPEND);
43 NU_Create_Task(&Task_1, "TASK 1", task_1, 0, NU_NULL, pointer, 1000, 10,
44 5, NU_PREEMPT, NU_START);

45 /* Create task 2. */
46 NU_Allocate_Memory(&System_Memory, &pointer, 1000, NU_NO_SUSPEND);
47 NU_Create_Task(&Task_2, "TASK 2", task_2, 0, NU_NULL, pointer, 1000,
48 10, 5, NU_PREEMPT, NU_START);

49 /* Create task 3. Note that task 4 uses the same instruction area. */
50 NU_Allocate_Memory(&System_Memory, &pointer, 1000, NU_NO_SUSPEND);
51 NU_Create_Task(&Task_3, "TASK 3", task_3_and_4, 0, NU_NULL, pointer,
52 1000, 5, 0, NU_PREEMPT, NU_START);

53 /* Create task 4. Note that task 3 uses the same instruction area.*/
54 NU_Allocate_Memory(&System_Memory, &pointer, 1000, NU_NO_SUSPEND);
55 NU_Create_Task(&Task_4, "TASK 4", task_3_and_4, 0, NU_NULL, pointer,
56 1000, 5, 0, NU_PREEMPT, NU_START);

57 /* Create task 5. */
58 NU_Allocate_Memory(&System_Memory, &pointer, 1000, NU_NO_SUSPEND);
59 NU_Create_Task(&Task_5, "TASK 5", task_5, 0, NU_NULL, pointer, 1000, 7, 0,
60 NU_PREEMPT, NU_START);

61 /* Create communication queue. */
62 NU_Allocate_Memory(&System_Memory, &pointer, 100*sizeof(UNSIGNED),
63 NU_NO_SUSPEND);
64 NU_Create_Queue(&Queue_0, "QUEUE 0", pointer, 100, NU_FIXED_SIZE, 1,
65 NU_FIFO);

66 /* Create synchronization semaphore. */
67 NU_Create_Semaphore(&Semaphore_0, "SEM 0", 1, NU_FIFO);

68 /* Create event flag group. */
69 NU_Create_Event_Group(&Event_Group_0, "EVGROUP0");
70 }

71 /* Define task 0. Task 0 increments the Task_Time variable every
72 18 clock ticks. Additionally, task 0 sets an event flag that
73 task 5 is waiting for, on each iteration of the loop. */

74 void task_0(UNSIGNED argc, VOID *argv)
75 {
76 STATUS status;

77 /* Access argc and argv just to avoid compilation warnings.*/
78 status = (STATUS) argc + (STATUS) argv;

```



```

79 /* Set the clock to 0. This clock ticks every 18 system timer ticks. */
80 Task_Time = 0;

81 while(1)
82 {

83 /* Sleep for 18 timer ticks. The value of the tick is programmable in
   IND.ASM and is relative to the speed of the target system. */
84 NU_Sleep(18);

85 /* Increment the time. */
86 Task_Time++;

87 /* Set an event flag to lift the suspension on task 5.*/
88 NU_Set_Events(&Event_Group_0, 1, NU_OR);
89 }
90 }

91 /* Define the queue sending task. Note that the only things that cause
92 this task to suspend are queue full conditions and the time slice
93 specified in the configuration file. */

94 void task_1(UNSIGNED argc, VOID *argv)
95 {
96 STATUS status;
97 UNSIGNED Send_Message;

98 /* Access argc and argv just to avoid compilation warnings. */
99 status = (STATUS) argc + (STATUS) argv;

100 /* Initialize the message counter. */
101 Task_1_messages_sent = 0;

102 /* Initialize the message contents. The receiver will
103 examine the message contents for errors. */
104 Send_Message = 0;

105 while(1)
106 {

107 /* Send the message to Queue_0, which task 2 reads from. Note
108 that if the destination queue fills up this task suspends until
109 room becomes available. */
110 status = NU_Send_To_Queue(&Queue_0, &Send_Message, 1,
111 NU_SUSPEND);

112 /* Determine if the message was sent successfully. */
113 if (status == NU_SUCCESS)
114 Task_1_messages_sent++;

115 /* Modify the contents of the next message to send. */
116 Send_Message++;
117 }
118 }

```



```

119  /* Define the queue receiving task. Note that the only things that
120  cause this task to suspend are queue empty conditions and the
121  time slice      specified in the configuration file.  */

122  void task_2(UNSIGNED argc, VOID *argv)
123  {
124      STATUS      status;
125      UNSIGNED Receive_Message;
126      UNSIGNED received_size;
127      UNSIGNED message_expected;

128  /* Access argc and argv just to avoid compilation warnings.  */
129  status = (STATUS) argc + (STATUS) argv;

130  /* Initialize the message counter.  */
131  Task_2_messages_received = 0;

132  /* Initialize the message error counter.  */
133  Task_2_invalid_messages = 0;

134  /* Initialize the message contents to expect.  */
135  message_expected = 0;

136  while(1)
137  {
138      /* Retrieve a message from Queue_0, which task 1 writes to.
139      Note that if the source queue is empty this task
140      suspends until something becomes available.  */
141      status = NU_Receive_From_Queue(&Queue_0, &Receive_Message, 1,
142      &received_size, NU_SUSPEND);

143  /* Determine if the message was received successfully.  */
144  if (status == NU_SUCCESS)
145      Task_2_messages_received++;

146  /* Check the contents of the message against what this task
147  is expecting.  */
148  if ((received_size != 1) ||
149      (Receive_Message != message_expected))
150      Task_2_invalid_messages++;

151  /* Modify the expected contents of the next message.  */
152  message_expected++;
153  }
154  }

155  /* Tasks 3 and 4 want a single resource. Once one of the tasks gets the
156  resource, it keeps it for 30 clock ticks before releasing it. During
157  this time the other task suspends waiting for the resource. Note that
158  both task 3 and 4 use the same instruction areas but have different
159  stacks.  */

160  void task_3_and_4(UNSIGNED argc, VOID *argv)
161  {
162      STATUS status;

163  /* Access argc and argv just to avoid compilation warnings.  */
164  status = (STATUS) argc + (STATUS) argv;

165  /* Loop to allocate and deallocate the resource.  */

```



```

166     while(1)
167     {
168         /* Allocate the resource.  Suspend until it becomes available.  */
169         status = NU_Obtain_Semaphore(&Semaphore_0, NU_SUSPEND);

170         /* If the status is successful, show that this task owns the
171            resource.  */
172         if (status ==  NU_SUCCESS)
173         {
174             Who_has_the_resource = NU_Current_Task_Pointer();

175             /* Sleep for 100 ticks to cause the other task to suspend on
176                the resource.  */
177             NU_Sleep(100);

178             /* Release the semaphore.  */
179             NU_Release_Semaphore(&Semaphore_0);
180         }
181     }
182 }

183 /* Define the task that waits for the event to be set by task 0.  */
184 void task_5(UNSIGNED argc, VOID *argv)
185 {
186     STATUS      status;
187     UNSIGNED    event_group;

188     /* Access argc and argv just to avoid compilation warnings.  */
189     status =  (STATUS) argc + (STATUS) argv;

190     /* Initialize the event detection counter.  */
191     Event_Detections = 0;

192     /* Continue this process forever.  */
193     while(1)
194     {
195         /* Wait for an event and consume it.  */
196         status = NU_Retrieve_Events(&Event_Group_0, 1, NU_OR_CONSUME,
197                                   &event_group, NU_SUSPEND);

198         /* If the status is okay, increment the counter.  */
199         if (status == NU_SUCCESS)
200             Event_Detections++;
201     }
202 }

```





Appendix

Nucleus PLUS Constants

This appendix contains all Nucleus PLUS constants referenced in Chapter 4 of this manual (Nucleus PLUS Services). Note that two listings are provided. The first listing is ordered alphabetically, the second numerically.

Nucleus PLUS Constants (Alphabetical Listing)

Name	Decimal Value	Hex Value
NU_ALLOCATE_MEMORY_ID	47	2F
NU_ALLOCATE_PARTITION_ID	43	2B
NU_AND	2	2
NU_AND_CONSUME	3	3
NU_BROADCAST_TO_MAILBOX_ID	16	10
NU_BROADCAST_TO_PIPE_ID	30	1E
NU_BROADCAST_TO_QUEUE_ID	23	17
NU_CHANGE_PREEMPTION_ID	11	B
NU_CHANGE_PRIORITY_ID	10	A
NU_CHANGE_TIME_SLICE_ID	65	41
NU_CONTROL_SIGNALS_ID	49	31
NU_CONTROL_TIMER_ID	58	3A
NU_CREATE_DRIVER_ID	60	3C
NU_CREATE_EVENT_GROUP_ID	37	25
NU_CREATE_HISR_ID	54	36
NU_CREATE_MAILBOX_ID	12	C
NU_CREATE_MEMORY_POOL_ID	45	2D
NU_CREATE_PARTITION_POOL_ID	41	29
NU_CREATE_PIPE_ID	25	19
NU_CREATE_QUEUE_ID	18	12
NU_CREATE_SEMAPHORE_ID	32	20
NU_CREATE_TASK_ID	2	2
NU_CREATE_TIMER_ID	56	38
NU_DEALLOCATE_MEMORY_ID	48	30
NU_DEALLOCATE_PARTITION_ID	44	2C
NU_DELETE_DRIVER_ID	61	3D
NU_DELETE_EVENT_GROUP_ID	38	26
NU_DELETE_HISR_ID	55	37
NU_DELETE_MAILBOX_ID	13	D
NU_DELETE_MEMORY_POOL_ID	46	2E
NU_DELETE_PARTITION_POOL_ID	42	2A
NU_DELETE_PIPE_ID	26	1A
NU_DELETE_QUEUE_ID	19	13
NU_DELETE_SEMAPHORE_ID	33	21
NU_DELETE_TASK_ID	3	3
NU_DELETE_TIMER_ID	57	39
NU_DISABLE_INTERRUPTS	Port Specific	Port Specific
NU_DISABLE_TIMER	4	4
NU_DRIVER_SUSPEND	10	A



Name	Decimal Value	Hex Value
NU_ENABLE_INTERRUPTS	Port Specific	Port Specific
NU_ENABLE_TIMER	5	5
NU_EVENT_SUSPEND	7	7
NU_FALSE	0	0
NU_FIFO	6	6
NU_FINISHED	11	B
NU_FIXED_SIZE	7	7
NU_MAILBOX_SUSPEND	3	3
NU_MEMORY_SUSPEND	9	9
NU_NO_PREEMPT	8	8
NU_NO_START	9	9
NU_NO_SUSPEND	0	0
NU_NULL	0	0
NU_OBTAIN_SEMAPHORE_ID	35	23
NU_OR	0	0
NU_OR_CONSUME	1	1
NU_PARTITION_SUSPEND	8	8
NU_PIPE_SUSPEND	5	5
NU_PREEMPT	10	A
NU_PRIORITY	11	B
NU_PURE_SUSPEND	1	1
NU_QUEUE_SUSPEND	4	4
NU_READY	0	0
NU_RECEIVE_FROM_MAILBOX_ID	17	11
NU_RECEIVE_FROM_PIPE_ID	31	1F
NU_RECEIVE_FROM_QUEUE_ID	24	18
NU_RECEIVE_SIGNALS_ID	50	32
NU_REGISTER_LISR_ID	53	35
NU_REGISTER_SIGNAL_HANDLER_ID	51	33
NU_RELEASE_SEMAPHORE_ID	36	24
NU_RELINQUISH_ID	8	8
NU_REQUEST_DRIVER_ID	62	3E
NU_RESET_MAILBOX_ID	14	E
NU_RESET_PIPE_ID	27	1B
NU_RESET_QUEUE_ID	20	14
NU_RESET_SEMAPHORE_ID	34	22
NU_RESET_TASK_ID	4	4
NU_RESET_TIMER_ID	59	3B
NU_RESUME_DRIVER_ID	63	3F
NU_RESUME_TASK_ID	6	6
NU_RETRIEVE_EVENTS_ID	40	28
NU_SEMAPHORE_SUSPEND	6	6
NU_SEND_SIGNALS_ID	52	34
NU_SEND_TO_FRONT_OF_QUEUE_ID	21	15

Name	Decimal Value	Hex Value
NU_SEND_TO_FRONT_OF_PIPE_ID	28	1C
NU_SEND_TO_MAILBOX_ID	15	F
NU_SEND_TO_PIPE_ID	29	1D
NU_SEND_TO_QUEUE_ID	22	16
NU_SET_EVENTS_ID	39	27
NU_SLEEP_ID	9	9
NU_SLEEP_SUSPEND	2	2
NU_START	12	C
NU_SUCCESS	0	0
NU_SUSPEND	0xFFFFFFFFFUL	FFFFFFFF
NU_SUSPEND_DRIVER_ID	64	40
NU_SUSPEND_TASK_ID	7	7
NU_TERMINATE_TASK_ID	5	5
NU_TERMINATED	12	C
NU_TRUE	1	1
NU_USER_ID	1	1
NU_VARIABLE_SIZE	13	D

Nucleus PLUS Constants (Numerical Listing)

Name	Decimal Value	Hex Value
NU_ENABLE_INTERRUPTS	Port Specific	Port Specific
NU_DISABLE_INTERRUPTS	Port Specific	Port Specific
NU_FALSE	0	0
NU_NO_SUSPEND	0	0
NU_NULL	0	0
NU_OR	0	0
NU_READY	0	0
NU_SUCCESS	0	0
NU_OR_CONSUME	1	1
NU_PURE_SUSPEND	1	1
NU_TRUE	1	1
NU_USER_ID	1	1
NU_AND	2	2
NU_CREATE_TASK_ID	2	2
NU_SLEEP_SUSPEND	2	2
NU_AND_CONSUME	3	3
NU_DELETE_TASK_ID	3	3
NU_MAILBOX_SUSPEND	3	3
NU_DISABLE_TIMER	4	4
NU_QUEUE_SUSPEND	4	4
NU_RESET_TASK_ID	4	4
NU_ENABLE_TIMER	5	5

Name	Decimal Value	Hex Value
NU_TERMINATE_TASK_ID	5	5
NU_FIFO	6	6
NU_RESUME_TASK_ID	6	6
NU_SEMAPHORE_SUSPEND	6	6
NU_EVENT_SUSPEND	7	7
NU_FIXED_SIZE	7	7
NU_SUSPEND_TASK_ID	7	7
NU_NO_PREEMPT	8	8
NU_PARTITION_SUSPEND	8	8
NU_RELINQUISH_ID	8	8
NU_MEMORY_SUSPEND	9	9
NU_NO_START	9	9
NU_SLEEP_ID	9	9
NU_CHANGE_PRIORITY_ID	10	A
NU_DRIVER_SUSPEND	10	A
NU_PREEMPT	10	A
NU_CHANGE_PREEMPTION_ID	11	B
NU_FINISHED	11	B
NU_PRIORITY	11	B
NU_CREATE_MAILBOX_ID	12	C
NU_START	12	C
NU_TERMINATED	12	C
NU_DELETE_MAILBOX_ID	13	D
NU_VARIABLE_SIZE	13	D
NU_RESET_MAILBOX_ID	14	E
NU_SEND_TO_MAILBOX_ID	15	F
NU_BROADCAST_TO_MAILBOX_ID	16	10
NU_RECEIVE_FROM_MAILBOX_ID	17	11
NU_CREATE_QUEUE_ID	18	12
NU_DELETE_QUEUE_ID	19	13
NU_RESET_QUEUE_ID	20	14
NU_SEND_TO_FRONT_OF_QUEUE_ID	21	15
NU_SEND_TO_QUEUE_ID	22	16
NU_BROADCAST_TO_QUEUE_ID	23	17
NU_RECEIVE_FROM_QUEUE_ID	24	18
NU_CREATE_PIPE_ID	25	19
NU_DELETE_PIPE_ID	26	1A
NU_RESET_PIPE_ID	27	1B
NU_SEND_TO_FRONT_OF_PIPE_ID	28	1C
NU_SEND_TO_PIPE_ID	29	1D
NU_BROADCAST_TO_PIPE_ID	30	1E
NU_RECEIVE_FROM_PIPE_ID	31	1F
NU_CREATE_SEMAPHORE_ID	32	20
NU_DELETE_SEMAPHORE_ID	33	21
NU_RESET_SEMAPHORE_ID	34	2

Name	Decimal Value	Hex Value
NU_RELEASE_SEMAPHORE_ID	36	24
NU_CREATE_EVENT_GROUP_ID	37	25
NU_DELETE_EVENT_GROUP_ID	38	26
NU_SET_EVENTS_ID	39	27
NU_RETRIEVE_EVENTS_ID	40	28
NU_CREATE_PARTITION_POOL_ID	41	29
NU_DELETE_PARTITION_POOL_ID	42	2A
NU_ALLOCATE_PARTITION_ID	43	2B
NU_DEALLOCATE_PARTITION_ID	44	2C
NU_CREATE_MEMORY_POOL_ID	45	2D
NU_DELETE_MEMORY_POOL_ID	46	2E
NU_ALLOCATE_MEMORY_ID	47	2F
NU_DEALLOCATE_MEMORY_ID	48	30
NU_CONTROL_SIGNALS_ID	49	31
NU_RECEIVE_SIGNALS_ID	50	32
NU_REGISTER_SIGNAL_HANDLER_ID	51	33
NU_SEND_SIGNALS_ID	52	34
NU_REGISTER_LISR_ID	53	35
NU_CREATE_HISR_ID	54	36
NU_DELETE_HISR_ID	55	37
NU_CREATE_TIMER_ID	56	38
NU_DELETE_TIMER_ID	57	39
NU_CONTROL_TIMER_ID	58	3A
NU_RESET_TIMER_ID	59	3B
NU_CREATE_DRIVER_ID	60	3C
NU_DELETE_DRIVER_ID	61	3D
NU_REQUEST_DRIVER_ID	62	3E
NU_RESUME_DRIVER_ID	63	3F
NU_SUSPEND_DRIVER_ID	64	40
NU_CHANGE_TIME_SLICE	65	41
NU_SUSPEND	0xFFFFFFFFFUL	FFFFFFFF



Appendix

Error Conditions

This appendix contains all Nucleus PLUS fatal system error constants, and error codes. If a fatal system error occurs, one of these constants is passed to the fatal error handling function `ERC_System_Error`.

If the system error is `NU_STACK_OVERFLOW`, the currently executing thread's stack is too small. The current thread can be identified by examination of the global variable `TCD_Current_Thread`. This contains the pointer to the current thread's control block.

If the system error is `NU_UNHANDLED_INTERRUPT`, an interrupt was received that does not have an associated LISR. The interrupt vector number that caused the system error is stored in the global variable `TCD_Unhandled_Interrupt`.

Nucleus PLUS Fatal System Errors

Name	Decimal Value	Hex Value
<code>NU_ERROR_CREATING_TIMER_HISR</code>	1	1
<code>NU_ERROR_CREATING_TIMER_TASK</code>	2	2
<code>NU_STACK_OVERFLOW</code>	3	3
<code>NU_UNHANDLED_INTERRUPT</code>	4	4

Nucleus PLUS Error Codes

Name	Decimal Value	Hex Value
<code>NU_END_OF_LOG</code>	-1	FFFFFFF
<code>NU_GROUP_DELETED</code>	-2	FFFFFFFE
<code>NU_INVALID_DELETE</code>	-3	FFFFFFFD
<code>NU_INVALID_DRIVER</code>	-4	FFFFFFFC
<code>NU_INVALID_ENABLE</code>	-5	FFFFFFFB
<code>NU_INVALID_ENTRY</code>	-6	FFFFFFFA
<code>NU_INVALID_FUNCTION</code>	-7	FFFFFFF9
<code>NU_INVALID_GROUP</code>	-8	FFFFFFF8
<code>NU_INVALID_HISR</code>	-9	FFFFFFF7
<code>NU_INVALID_MAILBOX</code>	-10	FFFFFFF6
<code>NU_INVALID_MEMORY</code>	-11	FFFFFFF5
<code>NU_INVALID_MESSAGE</code>	-12	FFFFFFF4
<code>NU_INVALID_OPERATION</code>	-13	FFFFFFF3
<code>NU_INVALID_PIPE</code>	-14	FFFFFFF2
<code>NU_INVALID_POINTER</code>	-15	FFFFFFF1
<code>NU_INVALID_POOL</code>	-16	FFFFFFF0



Name	Decimal Value	Hex Value
NU_INVALID_PREEMPT	-17	FFFFFFEF
NU_INVALID_PRIORITY	-18	FFFFFFEE
NU_INVALID_QUEUE	-19	FFFFFFED
NU_INVALID_RESUME	-20	FFFFFFEC
NU_INVALID_SEMAPHORE	-21	FFFFFFEB
NU_INVALID_SIZE	-22	FFFFFFEA
NU_INVALID_START	-23	FFFFFFE9
NU_INVALID_SUSPEND	-24	FFFFFFE8
NU_INVALID_TASK	-25	FFFFFFE7
NU_INVALID_TIMER	-26	FFFFFFE6
NU_INVALID_VECTOR	-27	FFFFFFE5
NU_MAILBOX_DELETED	-28	FFFFFFE4
NU_MAILBOX_EMPTY	-29	FFFFFFE3
NU_MAILBOX_FULL	-30	FFFFFFE2
NU_MAILBOX_RESET	-31	FFFFFFE1
NU_NO_MEMORY	-32	FFFFFFE0
NU_NO_MORE_LISRS	-33	FFFFFFDF
NU_NO_PARTITION	-34	FFFFFFDE
NU_NOT_DISABLED	-35	FFFFFFDD
NU_NOT_PRESENT	-36	FFFFFFDC
NU_NOT_REGISTERED	-37	FFFFFFDB
NU_NOT_TERMINATED	-38	FFFFFFDA
NU_PIPE_DELETED	-39	FFFFFFD9
NU_PIPE_EMPTY	-40	FFFFFFD8
NU_PIPE_FULL	-41	FFFFFFD7
NU_PIPE_RESET	-42	FFFFFFD6
NU_POOL_DELETED	-43	FFFFFFD5
NU_QUEUE_DELETED	-44	FFFFFFD4
NU_QUEUE_EMPTY	-45	FFFFFFD3
NU_QUEUE_FULL	-46	FFFFFFD2
NU_QUEUE_RESET	-47	FFFFFFD1
NU_SEMAPHORE_DELETED	-48	FFFFFFD0
NU_SEMAPHORE_RESET	-49	FFFFFFCF
NU_TIMEOUT	-50	FFFFFFCE
NU_UNAVAILABLE	-51	FFFFFFCD





Appendix

I/O Driver Request Structures

Nucleus PLUS I/O Driver Constants

Name	Decimal Value	Hex Value
NU_IO_ERROR	-1	FFFFFFFF
NU_INITIALIZE	1	1
NU_ASSIGN	2	2
NU_RELEASE	3	3
NU_INPUT	4	4
NU_OUTPUT	5	5
NU_STATUS	6	6
NU_TERMINATE	7	7

Nucleus PLUS I/O Driver C Structures

```

/* Define I/O driver request structures. */

struct NU_INITIALIZE_STRUCT
{
    VOID        *nu_io_address;    /* Base IO address */
    UNSIGNED    nu_logical_units;  /* Number of logical units */
    VOID        *nu_memory;        /* Generic memory pointer */
    INT         nu_vector;         /* Interrupt vector number */
};

struct NU_ASSIGN_STRUCT
{
    UNSIGNED    nu_logical_unit;    /* Logical unit number */
    INT         nu_assign_info;     /* Additional assign info */
};

struct NU_RELEASE_STRUCT
{
    UNSIGNED    nu_logical_unit;    /* Logical unit number */
    INT         nu_release_info;    /* Additional release info */
};

struct NU_INPUT_STRUCT
{
    UNSIGNED    nu_logical_unit;    /* Logical unit number */
    UNSIGNED    nu_offset;          /* Offset of input */
    UNSIGNED    nu_request_size;    /* Requested input size */
    UNSIGNED    nu_actual_size;     /* Actual input size */
    VOID        *nu_buffer_ptr;    /* Input buffer pointer */
};

struct NU_OUTPUT_STRUCT
{
    UNSIGNED    nu_logical_unit;    /* Logical unit number */
    UNSIGNED    nu_offset;          /* Offset of output */
    UNSIGNED    nu_request_size;    /* Requested output size */
    UNSIGNED    nu_actual_size;     /* Actual output size */
    VOID        *nu_buffer_ptr;    /* Output buffer pointer */
};

```



```

struct NU_STATUS_STRUCT
{
    UNSIGNED    nu_logical_unit;        /* Logical unit number */
    VOID        *nu_extra_status;      /* Additional status ptr */
};

struct NU_TERMINATE_STRUCT
{
    UNSIGNED    nu_logical_unit;        /* Logical unit number */
};

typedef struct NU_DRIVER_REQUEST_STRUCT
{
    INT          nu_function;           /* I/O request function */
    UNSIGNED     nu_timeout;            /* Timeout on request */
    STATUS       nu_status;             /* Status of request */
    UNSIGNED     nu_supplemental;        /* Supplemental information */
    VOID         *nu_supplemental_ptr; /* Supplemental info pointer */

    /* Define a union of all the different types of request
    structures. */
union NU_REQUEST_INFO_UNION
{
    struct NU_INITIALIZE_STRUCT    nu_initialize;
    struct NU_ASSIGN_STRUCT        nu_assign;
    struct NU_RELEASE_STRUCT       nu_release;
    struct NU_INPUT_STRUCT         nu_input;
    struct NU_OUTPUT_STRUCT        nu_output;
    struct NU_STATUS_STRUCT        nu_status;
    struct NU_TERMINATE_STRUCT     nu_terminate;
} nu_request_info;
} NU_DRIVER_REQUEST;

```





Appendix

Techniques for Conserving Memory

The Nucleus PLUS kernel was designed with an emphasis on speed, and on providing ample features and capacities for a broad range of applications. There are applications, however, where RAM space requirements must be minimized, even if this means some penalty in performance, or isolated reduction in functionality.

Our customers have discovered a number of techniques that can be used to reduce the RAM space required by Nucleus, depending on the specific user application. Some involve a trade-off in performance, others reduce the functionality of some feature. In each case, it is up to the user to determine if any of these techniques are appropriate for their application. We have investigated these techniques ourselves, some in considerable depth. We found no reason to expect any of them to cause problems, if applied as described.

The generic Nucleus PLUS code benefits from a refinement process resulting from extended usage by our customer base. Every version of Nucleus PLUS shares about 95% of its C code with every other version, regardless of target processor. Even a customer working with a recently developed new processor enjoys the advantage of using Nucleus PLUS code already proven by many previous customers. The techniques discussed in this Appendix have not necessarily enjoyed the benefit of this long-term refinement process.

Data Initialization

By default, `INC_Initialize` calls functions to initialize the data structures for every feature Nucleus offers. Even if a feature is not used, e.g. Mailboxes, its associated data structures will be created in RAM if its data initialization is done in `INC_Initialize`. These data structures include the List of Created Mailboxes, the Count of Total Mailboxes, and the Created Mailbox Protection Structure. No actual Mailboxes are created until the application makes a call to `NU_Create_Mailbox`. This pattern holds true for other features as well. To avoid creating unused data structures, remove from `INC_Initialize` the initialization for any features not used by the application.

NU_MAX_LISRS

The default size of this parameter, found in `NUCLEUS.H`, is the total number of interrupt vectors supported by the processor. It determines the size of two arrays, each of which must have an entry for every interrupt that can be used. The size may be reduced if some continuous group of interrupts at the end of the vector table is not used by the system. Since one of the arrays is accessed using the vector number as an index, the total must include even the unused interrupt vectors that come before the last one subject to use by the system.

Consider an example where a processor supports 256 interrupt sources, but the user's system will only have potential interrupt sources for, at most, 20 of them. If the usable interrupts are 0-14, 70-73, and 127, `NU_MAX_LISRS` can be reduced to 128.

TC_PRIORITIES

If an application does not require the full 256 separate task priorities, this parameter, in `TC_DEFS.H`, can be reduced accordingly. The priority levels available to tasks will then



be reduced. Only those in a range beginning at 0 and extending to one less than the new size value can be used.

HISR Stack Sharing

It is permissible for HISRs of the same priority to share a single stack. Simply give the same location for stack space every time a HISR of a given priority is created. Make sure the same *size* value is used in every case. Typically, an application needs to use at least two HISR stacks, one for application HISRs at a minimum of one HISR priority level, and one for the Timer HISR.

To get by with only one HISR stack for the entire system would involve using the same HISR priority for all application HISRs as for the Timer HISR, but it can be done. Use the global variables `_TMD_HISR_Stack_Ptr`, `_TMD_HISR_Stack_Size`, and `_TMD_HISR_Priority` for stack location, stack size, and HISR priority, respectively, in each application call to `NU_Create_HISR`.

TCD_Lowest_Set_Bit

The lookup table `TCD_Lowest_Set_Bit`, defined in `TCD.C`, is normally copied from ROM to RAM. It is accessed during task switching, and quicker access from RAM is desirable. This table is never changed, however. It can be made a 'const' type, to avoid copying it into RAM, and save 256 longwords there. The penalty is slower access to the table, in ROM, during task switches. Take advantage of this only if slower task switching can be tolerated.

Using a Smaller INT Option

There are a few platforms supported by Nucleus PLUS where an 'int' size less than the processor's default 'int' size is available as a compiler option. This offers potential savings in data space, but this feature cannot be used directly with Nucleus PLUS itself. The Nucleus PLUS INT data type is mapped to the compiler's 'int' type (in `NUCLEUS.H`). The processor-specific assembler files are written assuming an INT the size of the default 'int'.

Changing the size of INT would result in incompatibility between the assembly and the 'C' code in Nucleus PLUS. Application source code cannot be compiled with a different 'int' size than the Nucleus PLUS code. It is possible, however, to take advantage of this compiler feature for the user application, without actually mixing incompatible type sizes. To do so, map the Nucleus PLUS type INT, and any other Nucleus types originally mapped to 'int', to some other data type the same size as the compiler's default 'int' size. Nucleus PLUS will then be using no 'int' type data at all, and the compiler's optional smaller 'int' can be used for the rest of the application.

